Validierung von Software-Metriken für die funktionale Programmiersprache Erlang

am Beispiel eines Instant-Messaging-Systems

Diplomarbeit
zur Erlangung des akademischen Grades Diplom-Informatiker

eingereicht am Institut für Informatik
der Humboldt-Universität zu Berlin

von Daniel Warmuth

Betreuer:	Prof. Dr.-Ing. Beate Meffert
	Dr.-Ing. Robby Rochlitzer, ESYS GmbH
	Dr.-Ing. Frank Winkler
weiterer Gutachter:	Prof. Dr. sc. nat. Klaus Bothe
eingereicht am:	10. Mai 2012
verteidigt am:	14. Juni 2012

Inhaltsverzeichnis

Hinweise

Anstelle der im Titel verwendeten Bezeichnung „Metrik“ wird im Rest dieser Arbeit der treffendere Begriff „Maß“ verwendet (siehe Abschnitt 2.1.2.3, S. 22).

Quellennachweise werden wie folgt angegeben: [i:s], wobei i den Index der Quelle im Literaturverzeichnis und s die Seitenzahl in der Quelle angibt.

In dieser Arbeit verwendete Fach- oder Spezialbegriffe werden bei ihrer Definition oder Einführung *kursiv* gesetzt und bei ihrer späteren Verwendung in der Regel mit einem vorangestellten „·“ markiert. Diese Begriffe können im Index auf Seite 148 nachgeschlagen werden. Eigennamen von Sprachen, Produkten oder Maßen erscheinen in KAPITÄLCHEN, sonstige Hervorhebungen ***halbfett-kursiv***. Programmtext wird in `dicktengleicher` Schrift gesetzt.

Alle nachfolgend vorgestellten statistischen Ergebnisse sind signifikant gemäß dem jeweils *gewählten Signifikanzniveau* [94:116], soweit nicht anders angegeben. Das gewählte Signifikanzniveau wird mit α bezeichnet, das *beobachtete Signifikanzniveau* [94:120] mit p. Das gewählte Signifikanzniveau bezieht sich jeweils auf den ganzen darauffolgenden Abschnitt, falls nicht anders angegeben. Die Wahrscheinlichkeit des *Fehlers 2. Art* [94:118] wird gegebenfalls mit β bezeichnet. Die Einstufung von Korrelationen als „stark“, „schwach“ oder ähnliches ist bei fremden Ergebnissen von den jeweiligen Autoren übernommen.

1 Einleitung

Diese Arbeit beschäftigt sich mit der empirischen Überprüfung von Maßen für Eigenschaften von Software. Im ersten Kapitel wird das Problem der Softwaremessung und der notwendigen Validierung vorgestellt, ähnliche Untersuchungen anderer Autoren zusammengefasst und der Beitrag dieser Arbeit skizziert. Abschließend erfolgt ein Überblick über den Aufbau des restlichen Textes.

1.1 Problemstellung

Die Entwicklung eines umfangreichen Softwaresystems[1] ist ein komplexes Unterfangen: Anforderungen aus dem Anwendungsgebiet müssen spezifiziert, Struktur und Verhalten der Software müssen modelliert, implementiert und getestet werden. Das Gebiet der *Software-Technik* (engl. *Software Engineering*) befasst sich damit, Methoden zu entwickeln, um diese Prozesse möglichst effektiv und effizient beherrschen zu können. Ziel ist es, durch Entwicklungsprozesse hoher Qualität Software hoher Qualität zu entwickeln. Die Erreichung dieses Ziels zu quantifizieren und objektiv zu überprüfen, ist Gegenstand der ·Softwaremessung (engl. *Software Measurement*).

Die Maße der Softwaremessung sollen bestimmte Eigenschaften der Entwicklungsprozesse und -produkte erfassen, die für die erfolgreiche Entwicklung der Software relevant sind. Insbesondere wird versucht, von internen Eigenschaften, die vor dem und unabhängig vom realen Einsatz erfasst werden können, auf externe Eigenschaften schließen zu können, die sich erst zu einem späteren Zeitpunkt in der Interaktion mit der Umwelt manifestieren (siehe Abb. 1.1 auf der nächsten Seite). Bei implementiertem Programmcode interessiert etwa, wie leicht oder schwer der

[1] „System“, „Produkt“, „Software“ und „Programm“ werden im Folgenden synonym benutzt.

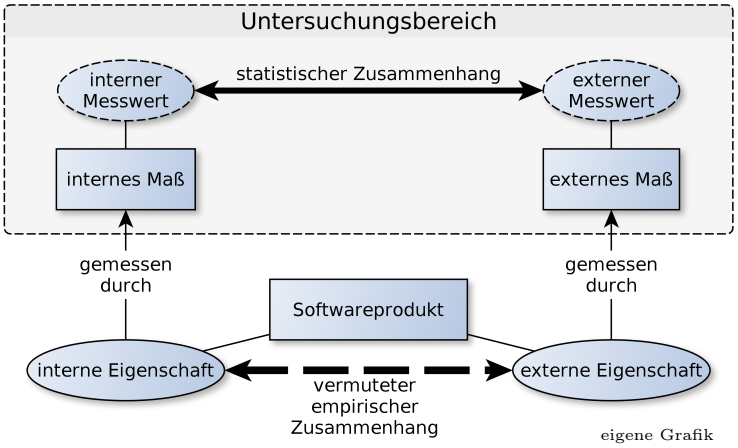


Abbildung 1.1 – Softwaremaße sollen interne und externe Eigenschaften insbesondere von Softwareprodukten erfassen. Von der Beziehung zwischen den Messwerten soll auf die Beziehung der empirischen Eigenschaften geschlossen werden können. Die Zulässigkeit dieser Schlussfolgerungen wird durch Validierung überprüft.

Code für Softwareentwickler zu verstehen ist; wie anfällig der Code für das Auftreten von Fehlern ist und wie aufwändig es ist, den Code zu testen und Fehler zu beheben. Verlässliche Aussagen über diese Eigenschaften helfen, den Entwicklungsprozess zu kontrollieren und zu planen. Daher ist es notwendig zu überprüfen, welche Maße Aussagen über welche Eigenschaften erlauben. Bei dieser ·Validierung müssen auch die verwendeten Entwicklungsmethoden berücksichtigt werden, da auch sie die zu messenden Eigenschaften beeinflussen. Ein Faktor dabei ist die verwendete Programmiersprache. Während viele Untersuchungen für imperative, das heißt prozedurale oder objekt-orientierte, Programmiersprachen existieren, gibt es nur wenige für funktionale Programmiersprachen [80:34]. Daher erfolgt in dieser Arbeit eine ·empirische Validierung ausgewählter Maße an Hand eines umfangreichen Softwaresystems, das in der Sprache Erlang implementiert ist. Dazu werden Zusammenhänge der erhobenen Messwerte mit der Menge der bekannten Fehler des betrachteten Systems untersucht.

1.2 Verwandte Arbeiten

Im Bereich funktionaler Programmiersprachen gibt es nach intensiver Recherche deutsch- und englischsprachiger Veröffentlichungen nur folgende empirische Unter-

suchungen zu Softwaremaßen: die Dissertationen von Berg [6] und von Ryder [80], sowie einen Artikel von Király und Kitlei [53].[2][3] Diese werden nun kurz vorgestellt.

1.2.0.1 Berg – Zusammenhänge mit Lesbarkeit und Verständlichkeit

Ausgehend von der Frage, ob Studierende, die funktionale Programmierung lernen, „bessere“ Programme schreiben als bei imperativer Programmierung, führt Berg [6] drei Experimente zur Untersuchung von Maßen für Lesbarkeit und Verständlichkeit durch. Betrachtet werden kleine Programme, die im Rahmen der Experimente von Studierenden nach einem einsemestrigen Programmierkurs geschrieben wurden.

Im ersten Experiment [6:38ff.] soll die Behauptung überprüft werden, funktionale Programmiersprachen führten im Vergleich zu imperativen Sprachen zu besser lesbaren Programmen [6:34]. Dazu werden Beziehungen der Maße ·zyklomatische Komplexität und *Programmier-Aufwand*[4] zur Lesbarkeit von Programmen der funktionalen Programmiersprache MIRANDA im Vergleich zu PASCAL-Programmen untersucht. Die Programme werden von Experten in eine Lesbarkeits-Rangfolge gestellt. Bei den PASCAL-Programmen ergeben sich für beide Maße hohe Korrelationen mit dieser Rangfolge,[5] für die MIRANDA-Programme keine signifikanten Korrelationen.[6] Die Aussagefähigkeit des Ergebnisses ist auf Grund des kleinen Stichprobenumfangs von neun beziehungsweise acht Programmen und der größeren Uneinigkeit der Experten bei der Lesbarkeitsbewertung[7] eingeschränkt. Die Hypothese, dass die beiden Maße zur Einschätzung der Lesbarkeit von MIRANDA-Programmen geeignet sind, wird nicht bestätigt.

Im zweiten Experiment [6:103ff.] betrachtet Berg Funktionstypausdrücke der Sprache MIRANDA, analog zu $f : \mathbb{N} \to \mathbb{R}$ aus der Mathematik. Er untersucht die

[2] Die Untersuchungen aus Ryder [80] wurden erneut in Ryder und Thompson [81] veröffentlicht.
[3] Király und Kitlei [53:280] verweisen auf Ryder [80], Ryder und Thompson [81] und Berg [6]. Ryder [80:40] wiederum nennt Berg [6] als einzige seinerzeit auffindbare Arbeit über Softwaremessung im Bereich funktionaler Programmierung.
[4] Nach Halstead [40:46ff.].
[5] Rangkorrelationen nach Spearman: $0,58$ (zyklomatische Komplexität) beziehungsweise $0,90$ (Programmier-Aufwand); α nicht angegeben, aber anscheinend $= 0,05$; $p \approx 0,05$ beziehungsweise $p \approx 0,00$.
[6] Rangkorrelationen: $0,56$ beziehungsweise $0,38$; $p \approx 0,07$ beziehungsweise $p \approx 0,18$.
[7] Kendalls Konkordanzkoeffizient: $0,50$ für Miranda gegenüber $0,74$ für Pascal; $p \approx 0,00$.

Beziehungen zwischen einem eigens definierten Maß [6:102] (im Folgenden m genannt) und der Zeit, die Versuchspersonen benötigen, um solche Ausdrücke zu verstehen. Das „Verstehen“ wird dadurch operationalisiert, dass die Versuchspersonen zu jedem Typausdruck eine beliebige passende Funktion definieren müssen. Insgesamt werden 16 Versuchspersonen jeweils die selben 40 Ausdrücke vorgelegt. Für m wurden sieben (Un)Gleichungen als Axiome festgelegt, die für die Rangordnung grundlegender Typausdrücke gelten sollen, das heißt für Basistypen sowie Listen, Tupel und Funktionen von Basistypen. So soll etwa eine Liste von Zahlen als schwerer verständlich eingeordnet werden als Zahlen selbst. Im ersten Teil des Experiments werden zunächst Bearbeitungszeiten für beide Seiten dieser (Un)Gleichungen ermittelt. Es werden folgende Hypothesen überprüft: Für die Ungleichungen sollen die Bearbeitungszeiten der beiden Seiten signifikant verschieden sein, für die Gleichungen soll die Differenz nicht signifikant sein. Etwa die Hälfte der Hypothesen wird bestätigt, für ein Gleichheits-Axiom ergibt sich wider Erwarten eine signifikante Differenz, für den Rest der Axiome sind die Differenzen konsistent, aber nicht signifikant.[8] Im zweiten Teil des Experiments wird die Rangordnung gemäß m mit derjenigen der gemessenen Zeiten verglichen. Die gefundene Rangkorrelation[9] lässt sich nicht bewerten, da Berg hier leider kein Signifikanzniveau angibt, ebenso bei einer Wiederholung dieses Experiments [6:111ff.,128ff.].[10]

Mit dem dritten Experiment [6:147ff.] soll untersucht werden, ob strukturierte Programme tatsächlich leichter und sicherer verständlich sind als unstrukturierte, wie es verschiedene Stilregeln nahelegen [6:142f.].[11] 94 Versuchspersonen werden je sechs Funktionen vorgelegt, die sich in Größe und Strukturiertheit unterscheiden: klein, mittel, groß beziehungsweise strukturiert, unstrukturiert. Die Versuchspersonen sollen zu jeder Funktion und einer gegebenen Eingabe das Ergebnis der Funktion angeben. Erfasst wird die benötigte Zeit und ob die Antwort korrekt ist. Es stellt sich heraus, dass Antworten bei strukturierten Funktionen schneller[12] und öfter korrekt[13] sind; dass bei größeren Funktionen langsamer[14] geantwortet wird;

[8] Angewendet wird der Fisher-t-Test; $\alpha = 0,05$ [6:105].
[9] Spearman-Korrelationskoeffizient von $0,59$.
[10] Dort: Pearson-Korrelationskoeffizient der Werte des selbstdefinierten Maßes und der gemessenen Zeiten von $0,80$, Spearman-Korrelationskoeffizient von $0,74$.
[11] Einzelheiten zur Bedeutung von ·strukturiert und ·unstrukturiert finden sich in Abschnitt 2.2.3.1, S. 30.
[12] Verwendet wird die F-Statistik [4:68ff.]; $p = 0,000$.
[13] Verwendet wird die Q-Statistik nach Cochran [72:376f.]; $p = 0,000$.
[14] Verwendet wird die F-Statistik [4:68ff.]; $p = 0,000$.

und, überraschenderweise, dass unter den strukturierten Funktionen bei größeren öfter korrekte Antworten erfolgen als bei kleineren.[15] In Bezug auf Letzteres vermutet Berg, dass möglicherweise auf größere Funktionen mehr Sorgfalt verwandt wird [6:163].

1.2.0.2 Ryder – Zusammenhang mit der Anzahl der Änderungen

Ryder [80:91ff.] untersucht Zusammenhänge einer Reihe von Softwaremaßen mit der Anzahl von Korrekturen, die im Laufe der Entwicklung an Programmen vorgenommen wurden. Codeänderungen werden manuell danach klassifiziert, ob sie neue Funktionalität hinzufügen oder vorhandene korrigieren. Als Korrekturen werden solche Änderungen gezählt, die Fehler beheben oder die Struktur verbessern sollen. Die Anzahl der Korrekturen interpretiert Ryder als Maß der Fehlerträchtigkeit [80:91]. Auf alternative Interpretationen, wie zum Beispiel als Hinweis auf gute Wartbarkeit, geht er nicht ein. Softwaremaße, die positiv mit der Korrekturenanzahl korrelieren, interpretiert Ryder als Maße für die „subjektive Komplexität“ von Code, das heißt die Schwierigkeit, Code zu verstehen oder zu ändern [80:103].

Untersucht werden zwei Programme aus universitären Forschungsprojekten, die in der funktionalen Programmiersprache Haskell geschrieben wurden: ein Spiel namens Peg Solitaire und eine Vor-Version des Refactoring-Werkzeugs HaRe.[16] Peg Solitaire hat nur etwa 900 Zeilen Code und 150 Änderungen [80:94], was dazu führt, dass sich meist keine signifikanten Ergebnisse ergeben. HaRe hat in der untersuchten Version etwa 5000 Zeilen und 450 Änderungen.[17] [18] Ryder ermittelt für jede Funktion der Programme die Maximalwerte einer Reihe von Softwaremaßen und die Anzahl von Codekorrekturen über die gesamte Entwicklungszeit [81:40]. Für jedes Softwaremaß ergibt sich also eine Datenreihe mit den Maximalwerten aller Funktionen. Eine weitere Datenreihe enthält die Anzahlen von Korrekturen dieser Funktionen. Untersucht wird die lineare Korrelation dieser

[15] $p = 0,000$ bis $0,004$

[16] HaRe – The Haskell Refactorer: `http://www.cs.kent.ac.uk/projects/refactor-fp/hare.html`

[17] Nachfolgend vorgestellte Ergebnisse stammen vom Programm HaRe, falls nicht anders angegeben.

[18] An einem Korpus von 14 weiteren Programmen mit insgesamt etwa 60 000 Zeilen Code untersuchte Ryder [80:96f.] Zusammenhänge zwischen verschiedenen Softwaremaßen, ohne jedoch externe Merkmale (wie die Anzahl von Korrekturen) zu betrachten. Auf diese Untersuchung wird hier nicht weiter eingegangen, da es sich nicht um eine Validierung im hier betrachteten Sinn (Abschnitt 2.3, S. 32) handelt.

Datenreihen ($\alpha = 0,05$ [80:103]). Zu kritisieren ist, dass Ryder wiederholt aus statistisch deutlich nicht signifikanten Ergebnissen Schlüsse zieht [vgl. 80:115] oder sich über deren Abweichung von signifikanten Ergebnissen wundert [vgl. 80:112]. Solche Schlussfolgerungen werden hier nicht wiedergegeben.

Eine Reihe der betrachteten Maße hängt intuitiv mit der Größe des Programms zusammen, daher überrascht es nicht, dass sie auch positiv mit der Anzahl der Korrekturen korrelieren. Die stärksten Korrelationen ($r \geq 0,5$) haben folgende Maße: Distanz zwischen Deklaration und Verwendung von Bezeichnern,[19] Anzahl in Mustern deklarierter Variablen,[20] Größe von Mustern,[21] Verschachtelungstiefe,[22] Anzahl der Operanden und Operatoren,[23] Ausgangsgrad[24] von Funktionen.[25]

Schwächere Korrelationen ($0,25 < r < 0,5$) ergeben folgende Maße: Tiefe und Kanten-Knoten-Verhältnis des Aufrufgraphen,[26] [27] Tiefe des Aufrufsubgraphen einzelner Funktionen,[28] Anzahl der Datentyp-Konstruktoren in Mustern,[29] Anzahl der Platzhalter-Variablen in Mustern,[30] Größe starker Zusammenhangskomponenten im Aufrufgraphen (nur bei nicht-trivialer Rekursion),[31] Anzahl von Ausführungspfaden.[32]

Keine signifikante Korrelation mit Korrekturen ergibt sich bei Maßen für Rekursion [80:135ff.].[33] Ryder [80:144] führt dieses Ergebnis auf Besonderheiten der un-

[19] Die Distanz wird durch Zählung der Sichtbarkeitsbereiche zwischen der Deklaration und den Verwendungsorten gemessen. Summe der Einzelwerte für alle Verwendungsorte: $r = 0,632$, $p < 0,0001$, Maximum der Einzelwerte: $r = 0,6006$, $p < 0,0001$ [80:128], lineare Regression mit Summe und Maximalwert: $r = 0,6829$, $p < 0,0001$ [80:277]. Geringere Korrelationen ergeben sich bei Zählung der Deklarationen in diesen Sichtbarkeitsbereichen ($r = 0,546$, $p < 0,0001$ für die Summe), der Anzahl dazwischenliegender Zeilen ($r = 0,5334$, $p < 0,0001$ für Maximalwert) und der dazwischenliegenden Knoten im ·Parse-Baum ($r = 0,54$, $p < 0,0001$ für Maximalwert) [80:265]

[20] $r = 0,5927$, $p < 0,0001$ [80:112]

[21] $r = 0,5423$, $p < 0,0001$ [80:119]

[22] $r = 0,4208$ bis $r = 0,5692$, $p < 0,0001$ [80:118]

[23] $r = 0,5795$ beziehungsweise $r = 0,558$ [80:156], bei $p < 0,0001$ [80:267]

[24] Siehe Abschnitt 2.2.2, S. 29.

[25] $r = 0,5723$, $p < 0,0001$ [80:148]

[26] Siehe Abschnitt 2.2.2, S. 29.

[27] $r = 0,4932$, $r = 0,4258$, bei $p < 0,0001$ [80:150, 266]

[28] $r = 0,3285$, $p < 0,0001$ [80:150, 266]

[29] $r = 0,3645$, $p < 0,0001$ [80:114]

[30] $r = 0,3572$, $p < 0,0001$ [80:117]

[31] $r = 0,3446$, $p < 0,0001$ [80:147, 266] bei Peg Solitaire, das nicht-triviale Rekursion enthält; $r = 0,0699$, $p = 0,105$ bei HaRe, das nur triviale Rekursion enthält.

[32] $r = 0,286$ [80:155], $p < 0,0001$ [80:267]

[33] Einzige, sehr schwache Ausnahme, wenn $\alpha = 0,10$, ist das Binärprädikat Rekursivität: Peg

tersuchten Programme zurück, die im Vergleich zu einem größeren Programmkorpus nur wenige rekursive Funktionen enthalten, die zudem so geartet sind, dass verschiedene Maße äquivalent oder nicht anwendbar sind [80:140ff.]. Fast keine Korrelation ergibt sich für den Eingangsgrad[34] von Funktionen.[35]

Ryder stellt abschließend fest, dass seine Resultate nur eingeschränkt verlässlich sind, da sie nur auf zwei untersuchten Programmen basieren, und regt weitere Untersuchungen an [80:163, 258]. Die gefundenen Messwerte liegen überwiegend im unteren Wertebereich, was darauf schließen lässt, dass Obergrenzen für akzeptable Werte gefunden werden können [80:256f.], wofür Ryder allerdings die Zeit fehlte.

1.2.0.3 Király und Kitlei – Prototypischer Einsatz von RefactorErl zur Softwaremessung

Die Autoren Király und Kitlei gehören zur Projektgruppe, die das in dieser Arbeit verwendete Werkzeug RefactorErl (siehe Abschnitt 3.3, S. 57) entwickelt. Király und Kitlei [53:275ff.] vergleichen zwei Versionen von RefactorErl hinsichtlich verschiedener Maße und stellen fest, dass die spätere der beiden Versionen umfangreicher ist, was sich in höheren Werten entsprechender Maße äußert und darin, dass die Analyse der späteren Version deutlich länger dauert. Ihre Untersuchung dient vor allem der Demonstration ihres Werkzeugs zur Softwaremessung. Inhaltlich ist sie hier nicht relevant, da es sich nicht um eine Validierungsstudie handelt.

1.3 Beitrag dieser Arbeit

Die bisher veröffentlichten Arbeiten zur Validierung von Software-Maßen für funktionale Programmiersprachen stützen sich auf Daten von kleinen bis mittelgroßen experimentellen Softwareprojekten mit nicht mehr als einigen Tausend Zeilen Code. In der vorliegenden Arbeit wird daher ein größeres Softwareprojekt betrachtet,

Solitaire – $r = 0,1119$, $p = 0,0883$ [80:266].

[34] Siehe Abschnitt 2.2.2, S. 29.

[35] $r = 0,0842$, $p = 0,0507$ [80:148]

das mehrere Zehntausend Zeilen Code enthält und dauerhaft professionell eingesetzt wird: der weit verbreitete Kommunikationsserver EJABBERD, der in der funktionalen Programmiersprache ERLANG implementiert ist. Verschiedene Softwaremaße werden erhoben und Zusammenhänge mit externen Qualitätsmerkmalen untersucht, die sich aus einer Datenbank bekannter Probleme und aus Codeänderungen ableiten lassen. Auf dieser Grundlage werden Aussagen getroffen, inwiefern bestimmte Softwaremaße als Indikatoren für bestimmte Eigenschaften eines Programms verwendet werden können. Zudem werden Vergleichswerte ermittelt, die bei der Bewertung der Messwerte anderer Systeme helfen können.

1.4 Aufbau

Diese Arbeit ist in sieben Kapitel und vier Anhänge gegliedert.

Bisher wurden die Softwaremessung und die Notwendigkeit der Validierung motiviert und der Beitrag dieser Arbeit im Kontext verwandter Arbeiten zusammengefasst.

Im Kapitel 2, S. 15 werden die Aspekte der Softwarequalität vorgestellt. Als Grundlage der Softwaremessung werden wesentliche Konzepte der Messtheorie definiert. Für den Teilbereich der Produktmaße werden Ziele und Anwendungen im Software-Entwicklungsprozess erläutert und es wird beschrieben, wie Softwareprodukte zum Zweck der Messung modelliert werden. Methoden der Validierung von Software-Maßen werden vorgestellt und abschließend wesentliche Aspekte der funktionalen Programmierung und ihre Auswirkungen bei der Softwaremessung diskutiert.

In Kapitel 3, S. 48 wird das Analysewerkzeug REFACTORERL und eine Auswahl von Maßen interner und externer Qualitätsmerkmale vorgestellt.

Kapitel 4, S. 61 führt das untersuchte Softwareprodukt EJABBERD ein. Nach der Beschreibung des Untersuchungsaufbaus und der Art der untersuchten Daten werden Hypothesen bezüglich der Zusammenhänge der internen und externen Maße aufgestellt, die schließlich statistisch überprüft werden.

Im Kapitel 5, S. 102 werden die Aussagen zu den Hypothesen zusammenfassend diskutiert.

Kapitel 6, S. 108 ist eine Zusammenfassung der Arbeit, Kapitel 7, S. 109 diskutiert offen gebliebene Fragen und mögliche Erweiterungen.

Nach dem Abbildungs-, dem Tabellen- und dem Literaturverzeichnis folgen Anhänge: Eine Übersetzung der Software Measurement Ontology (Anhang A, S. 123), die Dokumentation der Mess- und Auswertungsumgebung (Anhang B, S. 126) sowie ergänzende Materialien und Tabellen zu verschiedenen Kapiteln (Anhänge C und D).

2 Grundlagen

Nachdem im vorhergehenden Kapitel die Softwaremessung motiviert und die vorliegende Arbeit eingeordnet wurde, wird nun erläutert, wie Softwaremessung zur Qualitätskontrolle von Software beiträgt. Aspekte von Software-Qualität, Grundlagen des Messens und der Softwaremessung werden eingeführt und die Modellierung von Softwareprodukten zum Zwecke ihrer Vermessung beschrieben. Vorgehensweisen zur Validierung von Software-Maßen werden diskutiert. Abschließend wird die funktionale Programmierung und ihre Besonderheiten in Bezug auf die Softwaremessung erläutert.

2.1 Softwaremessung

> "The use of software metrics reduces subjectivity in the assessment and control of software quality by providing a quantitative basis for making decisions about software quality."
>
> *IEEE Standard for a Software Quality Metrics Methodology* [46:iii]

Softwaremessung ist ein Teilgebiet der *Software-Technik*, das heißt der "application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software" [19:67]. Gegenstand der Softwaremessung sind quantitative Maße für Eigenschaften der Prozesse und Produkte der Softwareentwicklung. Dementsprechend unterscheidet man Prozessmaße und Produktmaße [57:212]: *Prozessmaße* erfassen Eigenschaften des Entwicklungsprozesses wie Produktivität oder Effizienz, *Produktmaße* bilden Eigenschaften des Softwareprodukts wie Größe, Struktur, Fehlergehalt, Laufzeitverhalten ab. Produktmaße können statisch oder dynamisch, in verschiedenen Phasen des Entwicklungsprozesses und auf verschiedenen

Abstraktionsebenen erfasst werden: statisch an Dokumenten der Spezifikations-, Design- und Implementierungsphase, dynamisch während der Ausführung. Im Folgenden werden ausschließlich statische Produktmaße betrachtet.

2.1.1 Softwarequalität

Der ISO-Standard 25010 definiert *Softwarequalität* als "the degree to which the system satisfies the stated and implied needs of its various stakeholders, and thus provides value" [49:2].[36] Bei der Gesamtqualität werden *Produkt-* und *Datenqualität* sowie *Benutzungsqualität* unterschieden. Produkt- und Datenqualität betreffen diejenigen Eigenschaften des Softwareprodukts und der verarbeiteten Daten, die unabhängig von ihrem Verhalten sind und auch als *interne Eigenschaften* bezeichnet werden [28:74]. Die Benutzungsqualität bezieht sich auf das Zusammenspiel der Software mit der Hardware und den Benutzern, weshalb auch von *externen Eigenschaften* gesprochen wird [28:74]. Abb. 2.1 auf der nächsten Seite stellt die Zusammenhänge dieser Bereiche dar.

Die Produktqualität, von deren Messung diese Arbeit handelt, wird als Hierarchie verschiedener Eigenschaften modelliert, wobei abstraktere Eigenschaften in konkretere Untereigenschaften aufgegliedert werden. Das Ziel ist, Eigenschaften der untersten Ebene direkt durch (sogenannte *interne*) Softwaremaße erfassen zu können. ISO 25010 beschreibt Produktqualität durch acht Eigenschaften mit 32 Untereigenschaften [49:3f,10ff.]: Functional Suitability, Reliability, Performance efficiency, Operability, Security, Compatibility, Maintainability, Portability (siehe Abb. 2.2 auf der nächsten Seite). Ein Beispiel für ein Modell, das diese Untereigenschaften schließlich direkt auf Softwaremaße abbildet, ist das SIG Maintainability Model [42], das sich auf den Faktor *Wartbarkeit* (engl. *Maintainability*) nach ISO 9126 [47] bezieht. Diesen definiert ISO [49:14] als "degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers", wobei gilt: "Modifications can include corrections, improvements or

[36] Der ISO/IEC-Standard 25010 ist der Nachfolger des ISO/IEC-Standards 9126, den er seit März 2011 ersetzt [49:v]. Die deutsche Version des ISO/IEC 9126 war die DIN-Norm 66272 [25:22ff.], die allerdings ersatzlos zurückgezogen wurde (siehe `http://www.beuth.de/de/norm/din-66272/2385241`). Eine deutsche Fassung des Leitfadens für die gesamte ISO-Standard-Reihe 250xx ist in Vorbereitung (Stand 9. Mai 2012, siehe `http://www.nia.din.de/projekte/DIN+ISO%2FIEC+25000/de/136443377.html`).

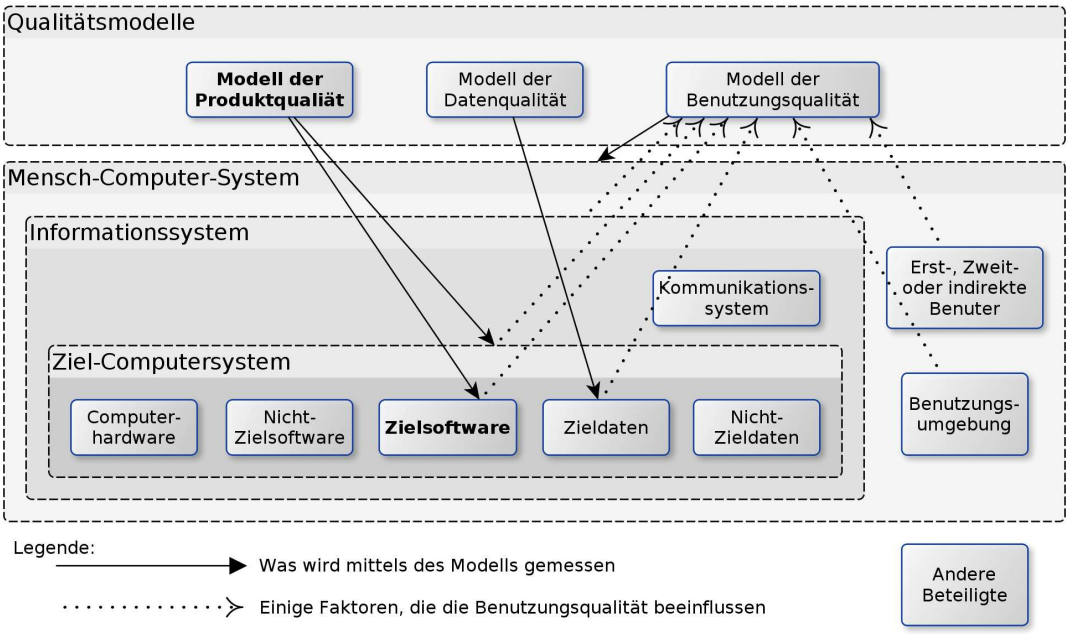


Abbildung 2.1 – Modelle für verschiedene Bereiche von Softwarequalität, und entsprechende Komponenten des Mensch-Computer-Systems nach [49:5]

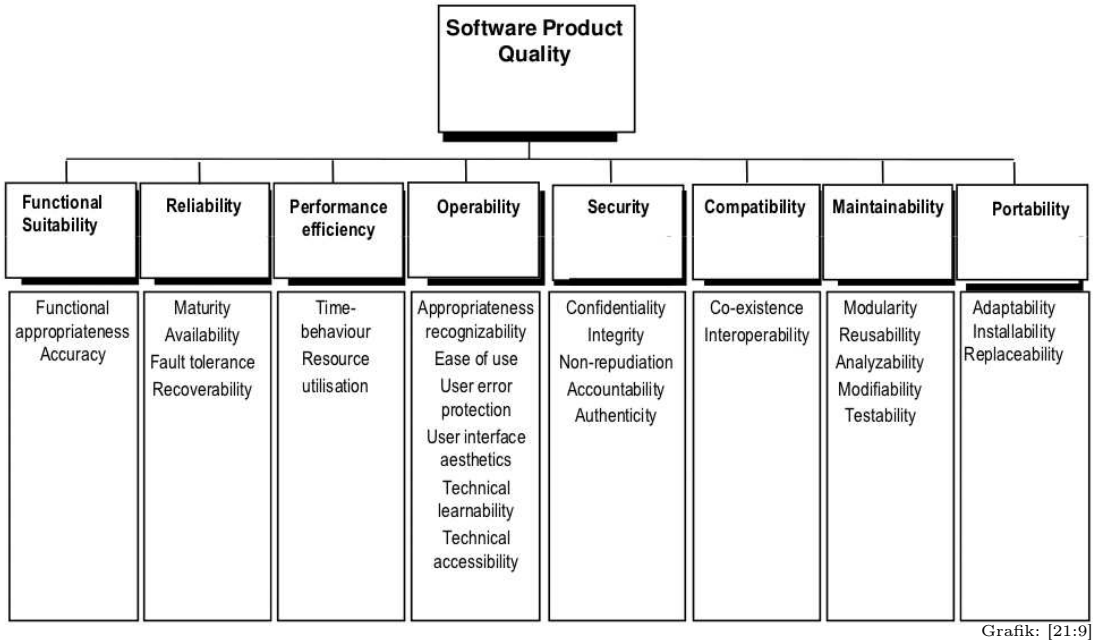
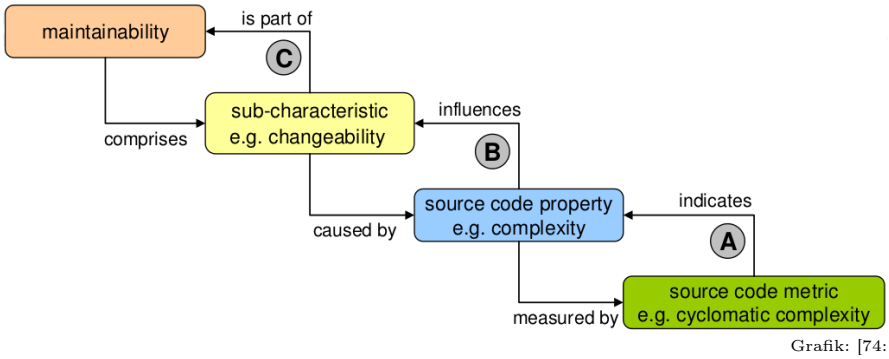


Abbildung 2.2 – Qualitätsmodell für Softwareprodukte nach ISO/IEC 25010 [49]

adaptation of the software to changes in environment, and in requirements and functional specifications."



Grafik: [74:12]

Abbildung 2.3 – Ausschnitt aus dem SIG Maintainability Model [42]

„Komplexität“

In der Literatur zur Softwaremessung ist oft von „Komplexität“ die Rede, ohne dass definiert wird, was darunter zu verstehen ist. Dieses vage Konzept wird als einzelne Eigenschaft behandelt und soll für die unterschiedlichsten Effekte verantwortlich sein: für das subjektive Empfinden, dass ein Stück Code schwer zu verstehen ist; für eine hohe Fehleranzahl im Code; für hohen Testaufwand; für Schwierigkeiten bei der Umstrukturierung von Software und so weiter. Damit einher geht das Streben, dieses ***komplexe*** Phänomen „Komplexität“ in einer ***einzigen*** Zahl ausdrücken zu wollen (ähnlich den Versuchen, „Intelligenz“ als einzelnen „Intelligenz-Quotienten“ auszudrücken [35]). Zuse [99:31] zitiert dazu Howatt und Baker [45]:

> “Measures of software properties should not be combined into a single-valued measure. One number cannot convey the information that a set of individual measures can; information is lost. We therefore propose that individual measures be made components of a vector of measures. This will provide complete information on each of the individual properties.”

Die Aspekte von Softwarequalität, die als „Komplexität“ bezeichnet werden, hängen im Unterschied zur Berechnungskomplexität neben objektiven Eigenschaften des Softwareprodukts auch von menschlichen Fähigkeiten ab. Ein Programm mit hoher Berechnungskomplexität kann leicht zu verstehen sein (zum Beispiel Bubblesort), das heißt niedrige „Komplexität“ haben; ein schwerer verständliches – „komplexeres“ – Programm kann hingegen effizienter sein (zum Beispiel Quicksort). Zuse [99:1] spricht daher auch von „psychologischer Komplexität“: „The true

meaning of the term software complexity is the difficulty to maintain, change and understand software. It deals with the psychological complexity of programs.“

2.1.2 Grundbegriffe der Softwaremessung

2.1.2.1 Messtheoretische Grundlagen

Messen wird in dieser Arbeit im Sinne der *repräsentationalen Messtheorie* [95:168] verstanden: „Das Messen ist eine Zuordnung von Zahlen zu Objekten oder Ereignissen, sofern diese Zuordnung eine homomorphe Abbildung eines empirischen Relativs in ein numerisches Relativ ist.“ [75:138] Im Folgenden werden die Begriffe *Relationensystem*[37] und *Maß* definiert. Eine weitergehende Einführung in diese Theorie des Messens findet sich in Bortz und Schuster [11:15f.].

Relationensystem Ein *Relationensystem* ist ein Tupel $(A, R_1, \ldots, R_n)$, wobei A eine nichtleere Menge von Objekten ist und R_i $(i = 1, \ldots, n)$ mehrstellige Relationen über A sind [99:40]. Wenn unter den Relationen abgeschlossene Binäroperationen auf den Elementen von A sind, werden diese separat mit $\circ_j$ $(j = 1, \ldots, m)$ bezeichnet.

Man unterscheidet empirische und formale Relationensysteme. Die Objekte eines *empirischen* Relationensystems sind zum Beispiel Programmtexte, die Relationen beispielsweise „ebenso verständlich“ oder „verständlicher“. Eine empirische Binäroperation ist das Zusammenfügen zweier Programmtexte. Die Objekte eines *formalen* Relationensystems sind üblicherweise Zahlen mit den algebraischen Relationen wie $\geq$, und Operationen wie Addition und Multiplikation.

Maß Es seien $\mathbf{E} = (E, R_1, \ldots, R_n, \circ_1, \ldots, \circ_m)$ ein empirisches Relationensystem, $\mathbf{F} = (F, S_1, \ldots, S_n, \bullet_1, \ldots, \bullet_m)$ ein formales Relationensystem, und μ eine Abbildung von E nach F. μ ist genau dann ein *Maß* (bezüglich $\mathbf{E}$ und $\mathbf{F}$), wenn es eine homomorphe Abbildung von E nach F ist [97:16]. Homomorph bedeutet in diesem Kontext, dass die Relationen der Messwerte den Relationen der empirischen Objekte entsprechen, das heißt für alle i, j und alle a, b, $a_{i_1}, \ldots, a_{i_k} \in E$

[37] In dieser Arbeit wird *Relationensystem* an Stelle des synonymen Begriffs „Relativ“ verwendet.

gilt [99:40f.] [vgl. 11:16]):

$$R_i(a_{i_1}, \ldots, a_{i_k}) \Leftrightarrow S_i(\mu(a_{i_1}), \ldots, \mu(a_{i_k}))$$

und

$$\mu(a \circ_j b) = \mu(a) \bullet_j \mu(b)$$

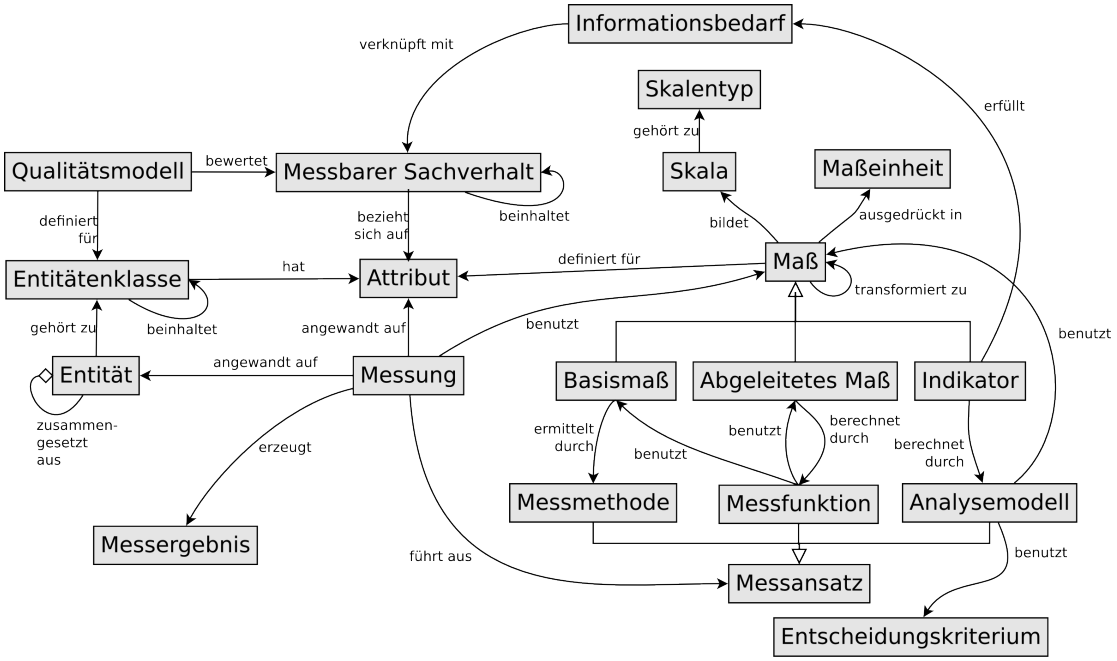
Skala Ist μ ein Maß bezüglich der Relationensysteme $\mathbf{E}$ und $\mathbf{F}$, wird das Tripel $(\mathbf{E}, \mathbf{F}, \mu)$ auch *Skala* genannt. Je nach Art der Relationen aus $\mathbf{E}$, die μ in $\mathbf{F}$ erhält, bildet es einen bestimmten der aus der Statistik bekannten *Skalentypen* (unter anderen Nominalskala, Rangskala, Intervallskala, Verhältnisskala und Absolutskala [89:6f.]). Zuse [99] leitet für eine Vielzahl von Softwaremaßen theoretisch den Typ der durch sie definierten Skala in Bezug auf die sogenannte „subjektive Komplexität“ her.

2.1.2.2 Ein einheitliches Vokabular

Auf Grund eines systematischen Vergleichs internationaler Standards und Forschungsveröffentlichungen zur Softwaremessung kommen García u. a. [32:631] zu dem Schluss, dass auf dem Gebiet der Softwaremessung „Terminologie, Prinzipien und Methoden immer noch definiert, gefestigt und vereinbart werden“[38] müssen. Sie fanden gravierende Fälle von Homo- und Synonymie sowie Unterschiede und Lücken auch bei grundlegenden Konzepten [32:635]. Auf einige solcher Probleme wird am Ende dieses Abschnitts eingegangen (Abschnitt 2.1.2.3, S. 22).

Durch Synthese und Vervollständigung der bestehenden Begrifflichkeiten entstand eine *informelle Ontologie* [56:274], die als strukturiertes, kontrolliertes Vokabular [56:280] zur Vereinheitlichung der Terminologie beitragen soll: die sogenannte Software Measurement Ontology [32:635ff.] [7:175ff.] (siehe Abb. 2.4 auf der nächsten Seite und Anhang A, S. 123 für eine eigene Übersetzung). Im Folgenden werden die in dieser Arbeit verwendeten Begriffe nach dieser Ontologie definiert, wobei die Definitionen für ·Maß und ·Skala durch die obigen ersetzt werden mussten, um mit der repräsentationalen Messtheorie konsistent zu sein.

[38] “. . . software measurement is currently in the phase in which terminology, principles, and methods are still being defined, consolidated, and agreed.”



eigene Grafik

Abbildung 2.4 – Angepasste Software Measurement Ontology [vgl. 7:181]

Messung Gesamtheit der Tätigkeiten, um mittels eines ·Messansatzes einen ·Messwert für ein bestimmtes Attribut eines ·Messobjekts zu ermitteln.

Messansatz Ein *Messansatz* ist eine Abfolge von Operationen, die darauf abzielen, den Wert eines Messergebnisses festzustellen. *Messmethode* und *Messfunktion* sind Arten von Messansätzen:

Messmethode Allgemein beschriebene logische Abfolge von Operationen, die angewendet werden, um ein Attribut in Bezug auf eine bestimmte ·Skala zu quantifizieren.

Messfunktion Algorithmus oder Berechnung, durch den/die mehrere ·Basismaße oder ·abgeleitete Maße verknüpft werden.

Messwert Die Zahl oder Kategorie, die durch eine ·Messung einem Attribut eines ·Messobjekts zugeordnet wird.

Messobjekt Empirisches Objekt, das durch ·Messung seiner Attribute charakterisiert werden soll.

Maß Homomorphe Abbildung μ der Menge der empirischen Objekte E in die Menge der formalen Objekte F. Für $e \in E$ heißt $\mu(e) \in F$ ·Messwert für e. Ein festgelegter ·Messansatz dient zur praktischen Umsetzung dieser theoretischen Abbildung. Man unterscheidet Basismaße und abgeleitete Maße:

Basismaß Ein ·Maß eines Attributs, welches auf keinem anderen ·Maß aufbaut und dessen ·Messansatz eine ·Messmethode ist.

Abgeleitetes Maß Ein ·Maß, das mittels einer ·Messfunktion als ·Messansatz aus anderen ·Basismaßen oder ·abgeleiteten Maßen gebildet wird.

Maße für ·interne Qualitätseigenschaften werden kurz *interne Maße* genannt, solche für ·externe Qualitätseigenschaften kurz *externe Maße*. Ein Beispiel für ein internes Maß ist die Anzahl der Zeilen eines Programms, während die Ausfallhäufigkeit ein externes Maß ist.

2.1.2.3 Mehrdeutige Begriffe

Begriff „Metrik“ In der Literatur zur Softwaremessung ist anstelle von ·Maßen oft von „Metriken“ die Rede (siehe zum Beispiel Balzert [5]). Dieser Begriff ist verwirrend, da *Metrik* in der Mathematik üblicherweise eine Abstandsfunktion für Punkte in einem Raum bezeichnet [33:766]. Messwerte in obigem Sinn bezeichnen jedoch keine Abstände, und nur unter bestimmten Bedingungen lassen sich Abstände zwischen Messwerten sinnvoll angeben. Zuse [99], Liggesmeyer [57] und andere weisen den Begriff „Metrik“ daher zurück und verwenden stattdessen den Begriff ·Maß.

Begriff „Maß“ Der ·Maß-Begriff der repräsentationalen Messtheorie unterscheidet sich vom Maß-Begriff der mathematischen Maßtheorie. Dort ist ein *Maß* eine Abbildung von Mengen einer σ-Algebra in die nichtnegativen reellen Zahlen, so dass unter anderem gilt, dass die Summe der Bilder disjunkter Mengen gleich dem Bild der Vereinigung dieser Mengen ist [3:17]. Für Wahrscheinlichkeitsmaße, die auf dieser Maßtheorie aufbauen, diskutieren Krantz u. a. [55:199ff.] deren Behandlung in Begriffen der repräsentationalen Messtheorie.

In [99:29] bezeichnet Zuse ***jede*** Abbildung μ der Menge der empirischen Objekte E in die Menge der formalen Objekte F als Maß – was problematisch wäre, da

somit nicht jede Anwendung eines „Maßes“ auch eine ·Messung wäre. Im Einklang mit seinen langjährigen Bemühungen, die Software-Messung auf eine solide messtheoretische Grundlage zu stellen, verwendet er in [97:16] und [98:3] jedoch die oben angegebene Definition.

Standardwerke zur repräsentationalen Messtheorie verwenden den Begriff „Maß“ (oder engl. „measure“) selten; Krantz u. a. [55], Orth [75] und Bortz und Schuster [11] sprechen von ·Skalen in obigem Sinn, ohne die homomorphe Abbildung, die zu einer ·Skala gehört, explizit als ·Maß zu bezeichnen.

Begriff „Skala“ „Skala“ wird als mathematischer Begriff oft verwendet, aber selten definiert. Nach Walz [93:39] bezeichnet der Begriff „Skala“ nur die *Bildmenge* eine ·Maßes im obigen Sinn. Eine „Skala“ wäre demnach nur eine Menge von Zahlen, die an sich nichts über bestimmte Relationen aussagen. In diesem Sinn wird „Skala“ und „Skalentyp“ auch in der Software Measurement Ontology definiert [32:636].

Oberflächlich betrachtet damit übereinstimmend bezeichnet DIN [24:26] eine „Größenwertskala“ als „Menge von Werten von Größen“. Allerdings sind dort Werte als „Zahlenwert ***und Referenz***“ [Hervorhebung hinzugefügt – d. Verf.] auf eine Maßeinheit, ein Messverfahren oder ein Referenzmaterial definiert [24:23]. Das bedeutet, es handelt sich nicht einfach um Zahlen, sondern um Zahlen mit einer bestimmten empirischen Bedeutung. Dies wird in der Definition einer ·Skala als Homomorphismus zwischen empirischem und formalem System nur deutlicher ausgedrückt.

2.1.3 Anwendungen von statischen Produktmaßen

> “You cannot understand the beauty of a painting by measuring its frame or understand the depth of a poem by counting the lines.”
>
> Marinescu und Lanza [60:46]

Softwaremessung ist eines von verschiedenen Mitteln zur Erfassung und Überprüfung von Softwarequalität. Andere Mittel sind die formale Spezifikation und Verifikation sowie das Testen (siehe Schlingloff [83:341ff.] für eine kurze Übersicht). Diese Mittel unterscheiden sich in ihrer Aussagekraft und Aufwändigkeit.

Mittels formaler Methoden können bestimmte Eigenschaften eines Programms ***bewiesen*** werden – allerdings sind diese Verfahren im Allgemeinen sehr aufwändig und schwer automatisierbar, so dass sie nur bei besonders wichtigen Systemen angewendet werden. Beim Testen kann der Aufwand durch die Wahl der Teststrategie und (halb-)automatische Testfallerzeugung reduziert werden, allerdings liefert es schwächere Aussagen als formale Methoden, denn: "Program testing can be used to show the presence of bugs, but never to show their absence!" [22:1]. Mit zunehmender Testüberdeckung sinkt allerdings die Wahrscheinlichkeit, dass Fehler unentdeckt bleiben.

Die Aussagen der Softwaremessung sind noch „schwächer", da sie oft nur auf ***mögliche*** Schwachstellen in einem Programm hinweisen. Darüberhinaus ist die Bedeutung von Softwaremaßen noch am wenigsten wissenschaftlich untersucht. Softwaremaße haben den Vorzug, dass sie auch mit geringem Aufwand und vollautomatisch ermittelt werden können.

Von statischen Produktmaßen für Implementierungsdokumente, wie sie in dieser Arbeit untersucht werden, wird meist erwartet zu quantifizieren, wie schwierig ein Codeabschnitt zu verstehen, zu verändern oder zu testen ist [80:11]. Balzert [5:232] betont, dass Softwaremaße mit Vorsicht zu betrachten seien. Da der Software-Entwicklungsprozess noch nicht vollständig verstanden sei, bildeten die Hypothesen über Zusammenhänge von Softwaremaßen und interessierenden Eigenschaften noch keine sichere Basis für quantitative Aussagen. Dieser Unsicherheit abzuhelfen, ist Anliegen der Validierung von Softwaremaßen.

Softwaremaße werden benutzt, um die gegenwärtige Qualität von Software zu erfassen und einzuordnen und die zukünftige Qualität vorherzusagen [84:412]. Aus der aktuellen oder vorausschauenden Qualitätsbewertung können Umstrukturierungsmaßnahmen abgeleitet werden [60].

2.1.3.1 Bewerten

Mittels Softwaremaßen können Qualitätsanforderungen an Software quantitativ festgelegt werden, um während und nach der Entwicklung ihre Veränderung überwachen und ihre Umsetzung überprüfen zu können [46:1]. Verschiedene Systeme oder Teile eines Systems können verglichen werden, um eine Auswahl zu treffen oder Entwicklungsaufwendungen zuzuweisen [86:326].

Softwaremaße abstrahieren jeweils von vielen Aspekten der Software, um dafür einen oder wenige Aspekte quantifizieren zu können. Ein einzelnes Softwaremaß erfasst daher nie ein Softwaresystem als Ganzes. Für umfassende Aussagen müssen mehrerere geeignet ausgewählte Maße ausgewertet werden [5:478]. Dabei können sinnvolle Zusammenfassungen mehrerer Maße (zum Beispiel die durchschnittliche Anzahl von Zeilen pro Klasse) Eigenschaften mitunter besser erkennbar machen als die Einzelwerte [60:23].

Bei der Anwendung von Softwaremaßen ist zu beachten, dass Ausreißer häufig vorkommen. Wenn eine Komponente aber in mehreren Maßen extreme Werte aufweist, sollte sie genauer untersucht und möglicherweise verändert werden [86:341] (siehe Abschnitt 2.1.3.3).

2.1.3.2 Vorhersagen

Softwaremaße können benutzt werden, um Eigenschaften vorherzusagen, die erst zu einem späteren Zeitpunkt messbar werden, wie zum Beispiel den Fehlergehalt von Programmen [57:231]. Auf Grund dieser Vorhersagen kann die zu erwartende Qualität frühzeitig bewertet werden, und vorbeugende Überprüfungs- oder Änderungsmaßnahmen veranlasst werden [84:412]. In der Literatur werden hauptsächlich Modelle untersucht, die die Fehlerträchtigkeit von Software vorhersagen sollen.[39] In dieser Arbeit werden Vorhersagemodelle nicht näher betrachtet, die Validierung von Softwaremaßen schafft allerdings eine Grundlage für die Entwicklung von Vorhersagemodellen aus diesen Maßen [39:3].

2.1.3.3 Umstrukturieren

Die Umstrukturierung eines Softwaresystems mit dem Ziel, die interne Struktur zu verbessern, ohne die externe Funktionalität zu ändern, heisst *Refactoring*[40] [30:xviii]. Einerseits existieren schematische Richtlinien für den Aufbau von

[39] Einen kritischen Überblick über solche Vorhersagemodelle bis einschließlich 1999 geben Fenton und Neil [27]; Hall u. a. [39] präsentieren eine systematische Literaturübersicht über Untersuchungen der Jahre 2000 bis 2010.

[40] Die Eindeutschung *Refaktorisieren* [30:xiii] ist eher unüblich.

Softwaresystemen – sogenannte *Entwurfsmuster* –, die zu höherer Softwarequalität führen sollen [31]. Andererseits werden Muster in der Struktur von Softwaresystemen beschrieben, die *vermieden* werden sollen, da sie die Softwarequalität negativ beeinflussten. Fowler [30:67ff.] spricht von *übel riechendem Code* (engl. ·bad smells in code), Brown u. a. [14] sprechen von „AntiPatterns“. Ein Beispiel ist, wenn der Großteil der Funktionalität eines objektorientierten Softwaresystems in einer einzigen Klasse zentralisiert ist, statt gleichmäßiger über Klassen mit klar abgegrenzten Zuständigkeiten verteilt zu sein (*Große Klasse* [30:71]). Dies gilt als hinderlich für die Wiederverwendbarkeit und Verständlichkeit dieser Klasse und des ganzen Systems [60:80].

Softwaremaße können zunächst nur zeigen, wie stark ein bestimmtes Attribut (zum Beispiel Wartbarkeit) von Software ausgeprägt ist. Meist ergibt sich daraus nicht direkt, wie die Software verbessert werden kann [vgl. 80:16]. Spinellis [86:331f.] weist beispielsweise darauf hin, dass sich der ·Wartbarkeitsindex auf triviale, aber nutzlose, Weise erhöhen („verbessern“) lässt, indem vor jede Funktion ein Kommentar eingefügt wird, der den Namen der Funktion und ihrer Parameter enthält. Dies würde jedoch nicht die Wartbarkeit des Codes verbessern, möglicherweise im Gegenteil verschlechtern. Spinellis [86] empfiehlt daher, nicht blindlings zu versuchen, Messwerte (hier des Wartbarkeitsindex) zu maximieren, sondern diese als Anhaltspunkte für potentiell problematische Codestellen zu benutzen, die dann zu begutachten sind. Liggesmeyer [57:239] warnt davor, Entscheidungen etwa über die Zerlegung von Modulen nur auf ein einzelnes Maß zu stützen, da dieses nur einen einzelnen Qualitätsaspekt vermisst.

Marinescu und Lanza [60] zeigen den Einsatz von Softwaremaßen zur automatischen Erkennung potentieller Schwachstellen im Design objektorientierter Softwaresysteme. Dazu stellen sie Regeln vor, die durch die Verknüpfung verschiedener Softwaremaße auf Muster schlechten Designs wie ·übel riechenden Code hinweisen sollen [60:49], als Ausgangspunkt für eine manuelle Überprüfung der betroffenen Komponenten [60:56f.]. Diese Regeln stellen Modelle dar, welche Aspekte von Softwarequalität durch welche Maße erfasst werden können. Abb. 2.5 auf der nächsten Seite zeigt die graphische Darstellung der Regel zur Erkennung des soeben beschriebenen Musters ·Große Klasse, das bei Marinescu und Lanza [60:80] „Gott-Klasse“ heißt.

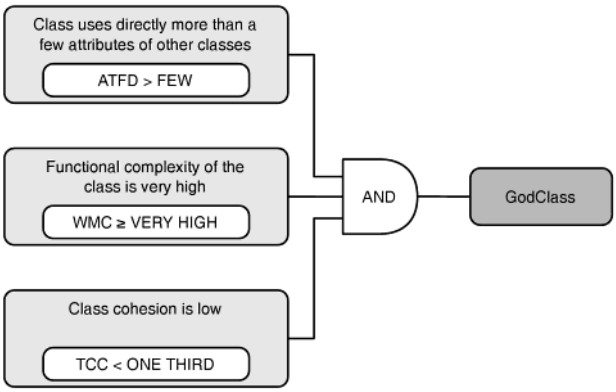


Abbildung 2.5 – Erkennungsregel für „Gott-Klasse“ bei Marinescu und Lanza [60:81], mit informellen Bedingungen und deren Abbildung auf drei Softwaremaße

2.1.4 Grenzen von statischen Produktmaßen

Die Aussagekraft von statischen Softwaremaßen ist durch verschiedene Faktoren begrenzt. Für dynamische Eigenschaften wie Geschwindigkeit oder Ressourcenbedarf können mittels statischer Analyse nur grobe Grenzen ermittelt werden. Andererseits können Methoden der statischen Analyse auch semantisch gehaltvollere Aussagen treffen als Softwaremaße. Zum Beispiel kann jeder Compiler tatsächliche Fehler erkennen und benennen. Softwaremaße sind abstrakter, ermöglichen dafür aber zusammenfassende Angaben über sehr große Systeme, um einen Überblick zu gewinnen, der mit detaillierten Angaben etwa über einzelne Fehler nicht möglich wäre. Zudem besteht bei Produktmaßen die Hoffnung, Schwachstellen von Programmen zu erkennen, ***bevor*** sich dort Fehler einstellen.

Bei der Interpretation von Produktmaßen muss beachtet werden, dass es neben den vermessenen viele weitere Einflüsse auf die Softwarequalität gibt, die auch berücksichtigt werden müssen (siehe Abschnitt 2.3.4, S. 37). Dazu gehören Eigenschaften anderer Produkte wie Spezifikations- und Designdokumente; des Entwicklungsprozesses, etwa verausgabte Arbeitszeit, Anzahl der Entwickler, verwendete Methoden; und schließlich der „menschliche Faktor“, wie Qualifikation und Motivation der Entwickler. Untersuchungen stellen beispielsweise fest, dass größere Programme proportional weniger Fehler enthalten [44:9f.] oder besser verstanden werden [6:163] und führen dies auf größere Sorgfalt der Entwickler zurück.

2.2 Modellierung von Programmen

Softwareprodukte werden zum Zweck der Messung auf verschiedene Weise modelliert. Die betrachteten Messobjekte sind also nicht die Softwareprodukte selbst, sondern ihre Modelle. Gigerenzer [34:60] spricht im Kontext der Psychologie von „Messung als Modellbildung“ und betont, dass die vermessenen Modelle nicht identisch mit den realen Objekten sind. Bei der Interpretation von Softwaremaßen ist zu berücksichtigen, von welchen empirischen Eigenschaften das jeweilige Modell abstrahiert.

Auf der niedrigsten Abstraktionsebene bestehen Programme in den meisten Programmiersprachen aus einer Zeichenfolge, durch die eine Folge von Symbolen der Programmiersprache kodiert wird. Einige Maße setzen direkt an der textuellen Form an, wobei sie vom semantischen Gehalt des Codes abstrahieren: Anzahl der Zeichen, Anzahl der Zeilen, Länge der Zeilen (siehe Abb. 2.6).

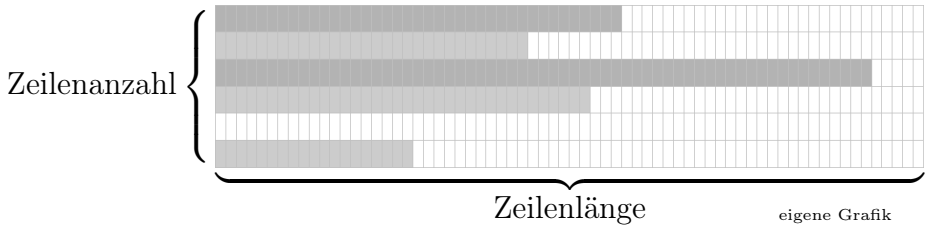


Abbildung 2.6 – Textform eines Programms

Andere Maße erfordern eine lexikalische Analyse, um die Symbole zu dekodieren, und oft auch eine syntaktische Analyse, um die grammatikalische Struktur festzustellen [2:6f.]; dazu gehört zum Beispiel die Anzahl der Kommentarzeilen.

Wieder andere Maße beziehen sich auf die Aufrufbeziehungen zwischen Funktionen, auf Import-Beziehungen zwischen Modulen oder auf den sogenannten Steuer- oder Kontrollfluss (Abschnitte 2.2.2 und 2.2.3). Deren Modellierung baut auf dem abstrakten Syntaxbaum auf, der nun vorgestellt wird.

2.2.1 Parse-Baum und (abstrakter) Syntaxbaum

Ein *Parse-Baum* (auch *konkreter Syntaxbaum* genannt) stellt die syntaktische Struktur eines Wortes einer Sprache dar, das heißt die Herleitung dieses Wortes mittels einer (kontextfreien) Grammatik [2:36]. Der Wurzelknoten entspricht dem

Startsymbol der Grammatik, jeder innere Knoten einem Nichtterminalsymbol und jedes Blatt einem Terminalsymbol oder dem leeren Wort ϵ.

Ein *abstrakter Syntaxbaum* (auch kurz *Syntaxbaum* oder *AST*) abstrahiert von syntaktischen Elementen, die für die Bedeutung eines Wortes nicht wesentlich sind [2:60]. So werden zum Beispiel Operatoren und Schlüsselwörter nicht als Blätter dargestellt, sondern als innere Knoten, mit ihren Operanden als Kindknoten [2:351].

2.2.2 Aufrufgraph

Ein (statischer) *Aufrufgraph* (engl. *call graph*) ist ein gerichteter Graph, der darstellt, welche Funktion welche Funktionen potentiell direkt aufrufen kann [41:4].

Jeder Knoten repräsentiert eine Funktion. Eine Kante (F_1, F_2) bedeutet, dass die vom Knoten F_1 repräsentierte Funktion die vom Knoten F_2 repräsentierte Funktion direkt aufrufen kann [41:12].

Analog dazu lässt sich ein gerichteter Graph definieren, der darstellt, welche Module welche anderen Module importieren, das heißt direkten Zugriff auf die darin enthaltenen Funktionen beanspruchen.

Die Anzahl der Kanten, die im Aufrufgraphen vom Knoten einer Funktion fortführen, kann kurz als *Ausgangsgrad* dieser Funktion bezeichnet werden, die Anzahl der Kanten, die im Aufrufgraphen zum Knoten einer Funktion hinführen, als *Eingangsgrad* dieser Funktion.

2.2.3 Kontrollflussgraph

Ein *Kontrollflussgraph* (engl. *control flow graph*,[41] kurz KFG) ist ein gerichteter Graph, der den potentiellen Programmablauf, den sogenannten *Kontrollfluss*, innerhalb einer Funktion darstellt [41:4].

[41] „control“ bedeutet hier eher „Steuerung“ (des Programmablaufs) als Kontrolle (im Sinne von Überwachung, Überprüfung). Daher ist in der Literatur (zum Beispiel in [2:80]) auch von „Steuerfluss“ die Rede, jedoch fast ausschließlich von „***Kontroll***flussgraph“. Zugunsten der Konsistenz wird daher in dieser Arbeit nur der Begriff „Kontrollfluss“ verwendet.

Jeder Knoten repräsentiert einen *Grundblock* (engl. *basic block*), das heißt „eine Folge fortlaufender Anweisungen, in die der Kontrollfluß am Anfang eintritt und die er am Ende verläßt, ohne daß er dazwischen anhält oder – außer am Ende – verzweigt“ [2:645]. Eine Kante (B_1, B_2) bedeutet, dass Block B_2 im Programmablauf direkt nach B_1 folgen kann [2:650], das heißt dass der Kontrollfluss von B_1 direkt auf B_2 übergehen kann [41:13].

Fenton, Whitty und Kaposi [29:146f.] definieren einen *KFG* formal als Tripel (G, a, z), wobei G ein endlicher gerichteter Graph und a und z ausgezeichnete Knoten von G sind. Für dieses Tripel muss gelten:

1. Alle Knoten außer z haben entweder Ausgangsgrad 1 (diese heißen *Prozedurknoten*) oder Ausgangsgrad 2 (diese heißen *Prädikatknoten*). Der Knoten z hat Ausgangsgrad 0.
2. Vom ausgezeichneten Knoten a (dem *Startknoten*) aus sind alle anderen Knoten von G erreichbar. Der ausgezeichnete Knoten z (der *Stopknoten*) ist von allen anderen Knoten aus erreichbar.

Programmkonstrukte, die zu höhergradigen Verzweigungen führen (zum Beispiel `switch` in der Sprache C) können als Kette von Prädikatknoten dargestellt werden. Alternativ kann die Definition und die darauf aufbauende Theorie leicht erweitert werden, so dass Prädikatknoten Ausgangsgrad ≥ 2 hätten [29:146].

2.2.3.1 Strukturiertheit

Nach Fenton, Whitty und Kaposi [29:154ff.] sind *strukturierte Programme* solche, die nur aus einer bestimmten Menge grundlegender Kontrollstrukturen aufgebaut sind. Fenton, Whitty und Kaposi [29:154] definieren eine *Kompositionsoperation*, mit der aus zwei KFGs F und G ein neuer KFG entsteht, indem ein Prozedurknoten x in F auf definierte Weise durch G „ersetzt“ wird. Dies entspricht zum Beispiel der Ersetzung eines Funktionsaufrufs im Programm durch die Definition der Funktion. Ein ähnliches Vorgehen liegt auch der Programmiermethode der ·schrittweisen Verfeinerung [26:7] zugrunde. Durch wiederholte Anwendung dieser Kompositionsoperation kann aus einer gegebenen Menge S von KFGs die Klasse der *S-Graphen* konstruiert werden – diese KFGs heißen *S-strukturiert* [29:155f.]. Whitty, Fenton und Kaposi [96] geben einen Überblick über die Entwicklung des

Begriffs ·strukturierte Programmierung. Angesichts verbreiteter Annahmen, strukturierte Programme seien leichter zu testen, zu debuggen, zu verstehen und somit zu ändern, betonen sie, dass die Überprüfung solcher Annahmen anerkannte Maße für Softwarequalität erfordert [96:55].

2.2.4 Fälle: Fehler und Ergänzungs-/Verbesserungswünsche

In einem *Fallbearbeitungssystem* werden während der Softwareentwicklung zu erledigende Aufgaben als sogenannte *Fälle* verwaltet, die Fehler, Verbesserungswünsche, Ergänzungswünsche oder andere Aufgaben repräsentieren. Jeder Fall hat einen Typ – FEHLER, VERBESSERUNG, ERGÄNZUNG, AUFGABE oder TEILAUFGABE. Für diese Untersuchung werden FEHLER als unerwünscht, VERBESSERUNGEN und ERGÄNZUNGEN als erwünscht und AUFGABEN als neutral angesehen werden.[42] Darüber hinaus hat ein Fall weitere Eigenschaften, insbesondere eine eindeutige Nummer, einen Erstellungszeitpunkt, zu dem er in das Fallbearbeitungssystem eingetragen wurde, und einen Lösungszeitpunkt, zu dem er als abgeschlossen markiert wurde. [43]

Ein Fall durchläuft im Fallbearbeitungssystem verschiedene Zustände (siehe Abb. 2.7 auf der nächsten Seite):

- Er wird „eröffnet“, das heißt in das System eingetragen.
- Er wurde einem Bearbeiter zugewiesen und befindet sich „in Bearbeitung“.
- Er wird als „gelöst“ markiert, weil die Bearbeitung abgeschlossen wurde. Dies kann bedeuten, dass ein betreffendes Problem gelöst wurde, aber auch zum Beispiel, dass der Fall als Duplikat eines anderen erkannt wurde.
- Er wird „geschlossen“, nachdem etwa ein anderer Entwickler die Lösung kontrolliert hat.
- Er wird „neu eröffnet“, weil sich beispielsweise eine vermeintliche Lösung als keine herausgestellt hat.

Diese Zustände können in unterschiedlicher Reihenfolge und auch wiederholt durchlaufen werden.

[42] Dies sind die Typen bei EJABBERD. In anderen Projekten können davon abweichende Typen definiert sein.

[43] Dies sind die Eigenschaften bei EJABBERD mit dem Fallbearbeitungssystem JIRA. In anderen Projekten und mit anderen Fallbearbeitungssystemen können davon abweichende Eigenschaften definiert werden.

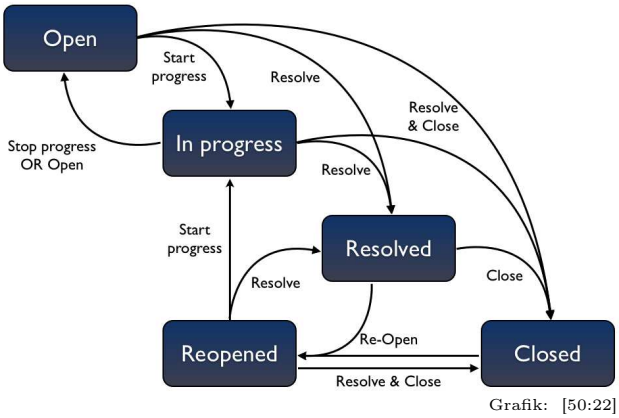


Abbildung 2.7 – Zustände eines ·Falls im Fallbearbeitungssystem JIRA

Anhand von Fällen können Maße externer Qualitätseigenschaften des entsprechenden Softwareprodukts ermittelt werden, etwa die Bearbeitungsgeschwindigkeit von Fällen (siehe Abschnitt 3.2, S. 55). Auf die für diese Untersuchung durchgeführte Analyse eines Fallbearbeitungssystems wird in Abschnitt 4.2.1.3, S. 65 näher eingegangen.

2.3 Validierung von Software-Maßen

> LOONQUAWL: "Forty-two! Is that all you've got to show for seven and a half million years' work?
> COMPUTER: I checked it very thoroughly, and that quite definitely is the answer. I think the problem, to be quite honest with you, is that you've never actually known what the question is."
>
> Douglas Adams [1:121]

Softwaremaße sollen bestimmte Attribute von Softwarequalität erfassen. Um festzustellen, ob Maße die Attribute, auf die sie sich beziehen, tatsächlich zutreffend wiedergeben, wurden verschiedene Kriterien und Methoden entwickelt, die zusammenfassend als *Validierung* bezeichnet werden. Der ISO-Standard 25010 definiert *Validierung* als "confirmation, through the provision of objective evidence, that the requirements for a specific intended use or application have been fulfilled" [49:20].

Man unterscheidet interne Validität, externe Validität und Konstruktvalidität. *Interne Validität* liegt vor, wenn das betrachtete Maß die Qualitätseigenschaft, die es messen soll, tatsächlich korrekt misst [97:407]. *Externe Validität* bedeutet, dass

das Maß auf bestimmte Weise mit einer ***anderen***, externen Qualitätseigenschaft zusammenhängt [97:407f.]. *Konstruktvalidität* bedeutet, dass die Definition und konkrete Umsetzung eines Maßes es erlauben, die zu messende Eigenschaft korrekt zu erfassen [65:16,25]. Aus dem Bezug auf bestimmte Benutzungsanforderungen folgt, dass die Validierung nicht endgültig abgeschlossen werden kann, sondern ein kontinuierlicher Prozess ist, der jeweils für veränderte Entwicklungsprozesse, Umgebungen und Projekte wiederholt werden muss [97:407,409f.].

Validierung kann *theoretisch*, das heißt mittels logischer Argumente, und *empirisch*, das heißt durch experimentelle Überprüfung, erfolgen [65:22] [65:17f.]. Meneely, Smith und Williams [65:24f.] weisen darauf hin, dass theoretische Validierung oft mit ·interner Validierung und empirische Validierung mit ·externer Validierung gleichgesetzt werden, jedoch eigentlich verschiedene Aspekte betreffen: „intern“ beziehungsweise „extern“ bezeichnet, ***was*** validiert wird, während „theoretisch“ beziehungsweise „empirisch“ angibt, ***wie*** validiert wird [65:24f.].

Nach welchen Kriterien die Validität von Maßen festgestellt werden soll, ist noch immer umstritten [65:1].Wie schon bei den Grundbegriffen der Softwaremessung (siehe Abschnitt 2.1.2.2, S. 20) gibt es eine Vielzahl konkurrierender Konzepte und Begriffe: Meneely, Smith und Williams [65:14ff.] haben in einer *systematischen Literaturübersicht* [54] 47 unterschiedliche Validierungskriterien zusammengetragen.

In den folgenden Unterabschnitten werden die von Meneely, Smith und Williams [65] zusammengestellten Validierungskriterien definiert (Nummerierung nach [65:14ff.]). Darunter sind insbesondere die Kriterien für externe, empirische Validität nach dem IEEE-Standard 1061 [46:11f.], die als unverbindlicher Anhang auch im ISO/IEC-Standard 25020 [48:10f.] enthalten sind: ·Assoziation, ·Trennschärfe, ·Eignung für Vorhersagen, ·Rangkonsistenz, ·Wiederholbarkeit, ·Parallele Veränderung. Abb. 2.8 auf der nächsten Seite zeigt die für diese Arbeit ausgewählten Kriterien, wobei die Kriterien nach IEEE 1061 und ISO/IEC 25020 hervorgehoben sind.

Eine Reihe der in der Literatur vorgeschlagenen Kriterien wurde für diese Arbeit verworfen. Diese Kriterien werden mit der Begründung ihres Ausschlusses in Anhang C.1, S. 130 aufgeführt.

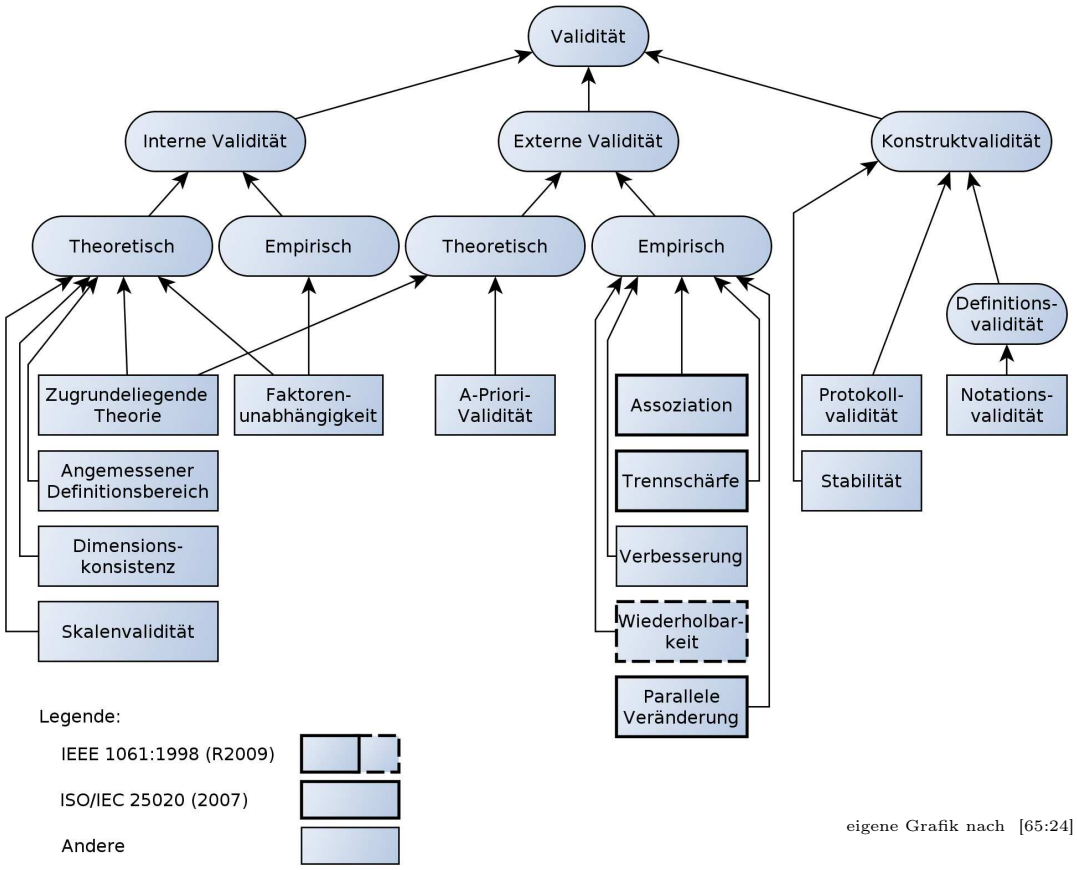


Abbildung 2.8 – Verschiedene in der Literatur vorkommende Validierungskriterien, ausgewählt nach Meneely, Smith und Williams [65]

2.3.1 Interne Validierung

In den folgenden Abschnitten werden einige theoretische und empirische Kriterien definiert, mit denen geprüft wird, ob ein Maß die zu messende Eigenschaft korrekt misst.

2.3.1.1 Theoretische interne Validierung

Angemessener Definitionsbereich, #3 Das Maß muss für alle real vorkommenden Ausprägungen einer Eigenschaft definiert sein und darf keine unerwarteten Lücken im Definitionsbereich aufweisen [65:15].

Dimensionskonsistenz, #14 Die ·Messfunktion eines ·abgeleiteten Maßes muss eine wissenschaftlich begründete und wohlverstandene Verknüpfung sein [65:17]. Das kann etwa bedeuten, verschiedene Maße nicht derart zu verknüpfen, dass aus der Verknüpfung keine Rückschlüsse mehr auf den Beitrag der einzelnen Komponenten gezogen werden können (siehe auch Abschnitt 2.1.1, S. 18).

Faktorenunabhängigkeit, #18 Die Komponenten ·abgeleiteter Maße sollen voneinander unabhängig sein [65:18]. Dies kann theoretisch überprüft werden, indem nach mehrfachem Auftreten von ·Basismaßen gesucht wird. Empirisch sollte aber auch die (unerwünschte) Korrelation der Komponenten überprüft werden.

Skalenvalidität, #40 Der ·Skalentyp eines Maßes soll explizit angegeben werden [65:21]. Vom Skalentyp hängt ab, welche Operationen, zum Beispiel statistische Berechnungen, auf Messwerte sinnvoll angewendet werden können.

Zugrundeliegende Theorie, #45 Siehe Abschnitt 2.3.2.1.

2.3.1.2 Empirische interne Validierung

Faktorenunabhängigkeit, #18 Siehe Abschnitt 2.3.1.1, S. 34.

2.3.2 Externe Validierung

Ob ein Maß mit einer externen Eigenschaft zusammenhängt, kann ebenso wie die interne Validierung sowohl theoretisch als auch empirisch überprüft werden. Es folgen einige der dabei angewandten Kriterien.

2.3.2.1 Theoretische externe Validierung

Zugrundeliegende Theorie und A-Priori-Validität #45/#1 Es soll eine dem Wissensstand des Anwendungsgebietes angemessene theoretische Grundlage für die Konstruktion des Maßes geben (*Zugrundeliegende Theorie*) [65:22]. Insbesondere soll der vermutete Zusammenhang zwischen Attributen ***vor*** der Überprüfung

postuliert werden, nicht erst im Nachhinein (*A-Priori-Validität*) [65:14]. So soll vermieden werden, dass einzelne zufällig signifikante Ergebnisse verallgemeinert werden.

2.3.2.2 Empirische externe Validierung

Assoziation, #5 Ein Maß soll unmittelbar mit einer externen Qualitätseigenschaft statistisch korreliert sein [65:15]. Genau genommen kann es sich nur um eine Korrelation mit einem ***Maß*** für eine externe Qualitätseigenschaft handeln.

Trennschärfe, #13 Es soll ein fester Schwellwert existieren, der Messobjekte niedriger Qualität von solchen hoher Qualität unterscheidet [65:17]. Ein solcher Schwellwert kann helfen festzustellen, welche Teile eines Softwaresystems etwa am intensivsten überarbeitet oder getestet werden müssen.

Verbesserungsvalidität, #19 Ein (neues) Maß soll in irgendeiner Weise eine Verbesserung gegenüber bisherigen Maßen darstellen [65:18], also etwa effizienter oder genauer sein. Eine übliche Anwendung dieses Kriteriums ist der Vergleich mit der Zeilenanzahl als „Referenzmaß“.

Wiederholbarkeit, #38 Ein Maß soll für verschiedene Projekte oder Stadien eines Projekts empirisch valide sein [65:21]. Diese Voraussetzung gilt für jegliche Forschungsergebnisse und macht deren Belastbarkeit davon abhängig, dass sie nachweislich nicht zufällig sind.

Parallele Veränderung, #43 Ein Maß soll sich im Zeitverlauf parallel zu einer externen Qualitätseigenschaft verändern [65:22]. Dies ist eine Abschwächung von ·Assoziation, die eine zeitliche Verschiebung korrespondierender Merkmalsausprägungen und Messwerte zulässt.

2.3.3 Konstruktvalidität

Definitionsvalidität, #12 Die Definition des Maßes muss klar und eindeutig sein, um eine genaue, objektive Messung zu ermöglichen [65:17]. Meneely, Smith und Williams [65:17] fordern zudem, dass sich das Messergebnis von dem anderer Maße unterscheiden soll. Letztere Forderung erscheint unnötig restriktiv – zum Beispiel kann eine Funktion ohne weiteres ebenso viele Zeilen wie Parameter haben, ohne dass dies zu Verwirrung führt. Ein Aspekt ist Notationsvalidität (#30), die vorliegt, wenn das Maß mathematisch präzise und konsistent notiert ist [65:19].

Werkzeugvalidität, #20 Das Messwerkzeug muss korrekt messen [65:18]. Dieses Kriterium erfordert die Verifikation des Messwerkzeugs und/oder ein Referenzwerkzeug zur Überprüfung der Messung.

Protokollvalidität, #35 Die Messung soll einem allgemein anerkannten „Messprotokoll“, das heißt ·Messansatz, folgen.

Stabilität, #41 Ein Maß soll unter gleichen Bedingungen die gleichen Messwerte ergeben [65:21]. Dies schließt Unabhängigkeit von subjektiven Urteilen ein und setzt ·Definitionsvalidität voraus.

2.3.4 Validierungsprozess

Im Validierungsprozess werden die internen und externen Validierungskriterien theoretisch und empirisch überprüft, um zu festzustellen, ob ein Maß tatsächlich eine homomorphe Abbildung der empirischen Objekte in die formalen Objekte darstellt. Abb. 2.9 auf der nächsten Seite veranschaulicht diesen Prozess in Bezug auf die Vermessung von Software***produkten*** (der Ablauf erfolgt in der Abbildung von unten nach oben).

Ein reales Softwareprodukt mit internen und externen Eigenschaften (in der Abbildung unten mittig) wird zunächst zum Zweck der Vermessung auf geeignete Weise modelliert (siehe Abschnitt 2.2, S. 28). Produktmaße erfassen nur die Eigenschaften dieser modellhaften Messobjekte, etwa Kontrollflussgraphen. Welche Art nützlicher Aussagen über das Softwareprodukt aus Produktmaßen abgeleitet

werden können, hängt also von der Art und Güte der Modellierung des Produkts durch die Messobjekte ab. Kontrollflussgraphen etwa abstrahieren von der Formatierung des Programmtextes, von Kommentaren, Dokumentation und vielem mehr, das für die Qualität des Produkts von Bedeutung ist. Diese Einschränkungen müssen bei der Anwendung von Produktmaßen berücksichtigt werden, um keine unzulässigen Aussagen über Eigenschaften abzuleiten, die im Messobjekt nicht modelliert sind.

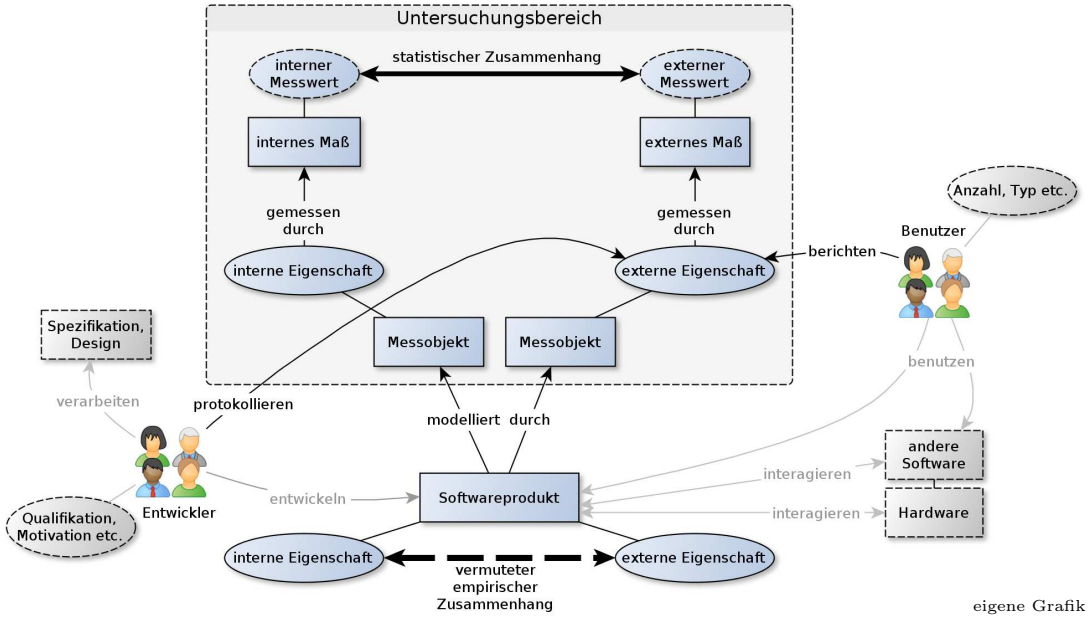


Abbildung 2.9 – Objekte und Akteure, die den Validierungsprozess beeinflussen

Je nachdem, ob interne oder externe Eigenschaften gemessen werden sollen, wird ein anderes Messobjekt erstellt: für interne Eigenschaften wie die Anzahl von Aufrufbeziehungen zwischen Funktionen etwa der erwähnte Kontrollflussgraph, für externe Eigenschaften wie die Bearbeitungseffizienz von Fehlern beispielsweise eine Sammlung von ·Fällen aus einem Fallbearbeitungssystem, die Fehler oder Schwachstellen des Softwareprodukts abbilden. Die Eigenschaften dieser Messobjekte werden dann ***stellvertretend*** für die entsprechenden Eigenschaften des Softwareprodukts durch interne oder externe Maße gemessen, woraus interne und externe Messwerte resultieren (in der Abbildung ganz oben). Bei der verbreitetesten Form der Validierung, der externen empirischen Validierung interner Maße, wird dann statistisch untersucht, welche Zusammenhänge zwischen diesen Messwerten bestehen. Unter der Voraussetzung, dass sowohl das interne als auch das

externe Maß intern valide sind, kann aus dem statistischen Zusammenhang auf die externe Validität des internen Maßes geschlossen werden, das heißt darauf, ob das interne Maß mit einer externen Eigenschaft zusammenhängt.

Es werden nun weitere Faktoren umrissen, die die Softwarequalität beeinflussen, gefolgt von einer Diskussion der Herausforderungen bei der Untersuchung von Softwarefehlern als Grundlage externer Eigenschaften.

2.3.4.1 Vielfältige Einflüsse auf die Softwarequalität

Die Qualität des Softwareprodukts wird von vielfältigen Faktoren beeinflusst ist, die in Softwaremaßen nur indirekt widergespiegelt werden. Diese Einflüsse sind in Abb. 2.9, S. 38 durch Pfeile mit dem Produkt und seinen Modellen verknüpft. Sie zerfallen in drei Bereiche: Einflüsse seitens der Softwareentwickler, seitens der Benutzer und Einflüsse durch Hardware und andere Software.

Die Entwickler der Software nehmen offensichtlich starken Einfluss auf deren Qualität. Wie sich dieser Einfluss gestaltet, hängt unter anderem von der fachlichen Qualifikation, vom Motivationsgrad und der Leistungsfähigkeit während des Entwicklungsprozesses ab. Mangelhaft ausgebildete Entwickler produzieren vermutlich Software niedrigerer Qualität als gut ausgebildete; höher motivierte Entwickler lassen möglicherweise größere Sorgfalt walten und produzieren so etwa weniger Fehler; Entwickler, die durch zu lange Arbeitszeiten oder andere Aufgaben stärker ausgelastet sind, erreichen beim untersuchten Produkt eher nicht ihre volle Produktivität. Zudem sind Entwickler auf Vorleistungen anderer Entwickler, wie Spezifikations- oder Entwurfsdokumente, angewiesen. Deren Qualität beeinflusst offenbar auch die Qualität des Endprodukts.

Die Betriebsumgebung des Softwareprodukts, die aus Hardware und anderer Software besteht, beeinflusst die Qualität, da das Softwareprodukt bestimmte Eigenschaften aufweisen muss, um mit der Umgebung interagieren zu können.

Auch die Benutzer üben einen Einfluss auf die Softwarequalität aus, indem sie durch Benutzung dessen Eigenschaften erleben und die Entwickler direkt oder indirekt zu Veränderungen am Produkt oder an zukünftigen Produkten anhalten. Näheres dazu im nächsten Abschnitt.

Alle diese Faktoren wirken als Hintergrundvariablen auf die Softwarequalität ein. Ziel der Validierung muss es daher sein, den Einfluss dieser Variablen bei der Bewertung von Softwaremaßen zu berücksichtigen. Dies stößt auf erhebliche Schwierigkeiten, da es schon eine seltene Gelegenheit ist, wenn zu einem Untersuchungsobjekt sowohl Quellcode als auch umfassende Fehleraufzeichnungen für einen langen Zeitraum verfügbar sind [44:2] – konkrete Informationen über Eigenschaften der Entwickler, Benutzer oder der Betriebsumgebung sind nur schwer zu erlangen.

2.3.4.2 Empirische Validierung in Bezug auf Softwarefehler

Da ein zentrales Anliegen bei der Softwareentwicklung die Vermeidung und gegebenfalls schnelle und effiziente Beseitigung von Softwarefehlern ist, werden bei empirischen Validierungsstudien meist externe Eigenschaften betrachtet, die mit Fehlern im Softwareprodukt verbunden sind.

Fehlerhafte Software, die zum Einsatz kommt, kann schädliche Auswirkungen haben und so noch höheren Aufwand zur Behebung der entstandenen Schäden verursachen, sei es der Absturz eines Flugzeugs wegen fehlerhafter Steuerungssoftware oder Datenverlust auf Grund eines Fehlers beim Speichern eines Dokuments. Softwarefehler müssen möglichst frühzeitig vermieden oder gegebenenfalls erkannt werden, weil der Aufwand für ihre Behebung mit fortschreitendem Entwicklungsprozess exponentiell ansteigt.

Dabei wird zwischen Fehlerursache, Fehlerzustand und Fehlerauswirkung unterschieden [83:333]. Die *Fehlerursache* liegt als Irrtum oder Störung außerhalb des Softwaresystems. Als *Fehlerzustand* bezeichnet man die interne Eigenschaft des Softwareprodukts, die unter Umständen zu fehlerhaftem Verhalten oder zum Ausfall der Software, zur *Fehlerauswirkung*, führt. Diese Eigenschaft kann permanent statisch vorliegen oder sich dynamisch während der Ausführung einstellen. Im Folgenden werden nur statische Fehlerzustände betrachtet, die kurz *Fehler* genannt werden. Nach Schlingloff [83:333] sind „Softwarefehler immer systematischer Natur und lassen sich auf Irrtümer bei der Konstruktion der Software zurückführen“. Von verschiedenen Softwaremaßen wie der ·zyklomatischen Komplexität wird angenommen, dass sie mit der Wahrscheinlichkeit solcher Irrtümer in Beziehung stehen.

Große Softwaresysteme enthalten wahrscheinlich immer mehr ·Fehler, als auf Grund beobachteter ·Fehlerauswirkungen bekannt sind. Die *bekannten Fehler* sind also nur eine Teilmenge aller ·Fehler. Von den bekannten Fehlern wird wiederum nur eine Teilmenge in potentiellen Datenquellen empirischer Untersuchungen vermerkt. Eine Art solcher Datenquellen sind ·Fallbearbeitungssysteme (siehe Abschnitt 2.2.4, S. 31).

Die Fälle in einem Fallbearbeitungssystem, die vom Typ FEHLER sind (siehe Abschnitt 2.2.4, S. 31), repräsentieren eine Stichprobe aus allen bekannten Fehlern. Diese Stichprobe ist zu einem bestimmten unbekannten Ausmaß fehlerhaft und verzerrt. Hooimeijer und Weimer [43:35] berichten etwa, dass zwischen 2003 und 2006 beim MOZILLA-Projekt 140 FEHLER-Fälle erstellt wurden, die sich auf den selben ·Fehler in einer Fortschrittsanzeige bezogen. Diese 140 FEHLER wurden nur nach und nach als Duplikate erkannt und markiert, so dass zu jedem beliebigen Zeitpunkt dieser einzelne Fehler im Fallbearbeitungssystem deutlich überrepräsentiert war. Die Qualität und Relevanz der FEHLER und anderer Fälle zu bestimmen, ist ein kompliziertes Problem für sich, das in empirischen Studien im Gebiet der Softwaretechnik jedoch selten angegangen wird [9:122].

Für die Untersuchung von Beziehungen zwischen internen Softwaremaßen und Fehlern müssen Fehler den betroffenen Programmkomponenten zugeordnet werden. Die Standardmethode, die auch in der vorliegenden Arbeit angewendet wird, ist laut Bird u. a. [9:125], die Einträge im Änderungsprotokoll von Versionsverwaltungssystemen wie CVS oder GIT nach den Bezeichnern von FEHLERN zu durchsuchen und einer Programmkomponente diejenigen FEHLER zuzuordnen, die in ihren Änderungseinträgen vermerkt sind. Mit dieser Methode können fast alle FEHLER zugeordnet werden, die in Änderungseinträgen vermerkt sind [9:125]. Jedoch sind oft nur ein Viertel bis die Hälfte aller FEHLER derart vermerkt. Laut Bird u. a. [9:129] ist es sehr aufwändig festzustellen, auf welche Weise diese Stichprobe der Fehler verzerrt ist – was es erlauben würde, Rückschlüsse auf die nicht verknüpften FEHLER zu ziehen.

Da Fehler oft nur als FEHLER registriert werden können, wenn es zu einer ·Fehlerauswirkung gekommen ist, hängt die Verteilung der FEHLER auf die Programmkomponenten nicht nur von der Verteilung tatsächlicher ·Fehler ab, sondern auch von der Ausführungsdauer und -häufigkeit sowie der Größe der Komponenten und vom Meldeverhalten der Entwickler und Benutzer. Mit zunehmender Lebenszeit

oder Ausführungsdauer einer Komponente steigt die Wahrscheinlichkeit, dass vorhandene Fehler auftreten und auch gemeldet werden. Mit zunehmender Größe einer Komponente hingegen ist anzunehmen, dass die Auftretenswahrscheinlichkeit jedes einzelnen Fehlers sinkt, da die Wahrscheinlichkeit der Ausführung jedes entsprechenden Programmabschnitts tendenziell sinkt [57:247]. Eine selten ausgeführte Komponente voller Fehler kann so als völlig FEHLER-frei erscheinen. Die Häufigkeitsverteilung der Nutzungsdauer und -intensität und daher auch deren Einfluss auf die Anzahl bekannter Fehler sind meist unbekannt [44:12]. Das für diese Arbeit entwickelte Verfahren zur Zuordnung von FEHLERN und anderen ·Fällen zu Programmkomponenten wird in Abschnitt 4.2.3, S. 69 näher beschrieben.

2.4 Funktionale Programmierung

> „Das zentrale Anliegen der funktionalen Sprachen ist, etwas von der Eleganz, Klarheit und Präzision der Mathematik in die Welt der Programmierung einfließen zu lassen."
>
> Peter Pepper [76:2]

Für die Entwicklung von Software gibt es verschiedene grundlegende Herangehensweisen, die Stile, Paradigmen oder Modelle genannt werden [77:892ff.]. Eine Unterscheidung ist die in ·imperative Programmierung und ·deskriptive Programmierung [12:6] (siehe Abb. 2.10, S. 45).[44] Zur imperativen Programmierung gehören die vorherrschenden Varianten *prozedurale Programmierung* und *objektorientierte Programmierung*, denen bekannte Sprachen wie PASCAL und C beziehungsweise C++ und JAVA zugerechnet werden. Bei diesem Ansatz bestehen Programme aus einer Reihe von Anweisungen, die vom Computer schrittweise ausgeführt werden, dabei explizit den Zustand des Computers verändern und so ein Ergebnis berechnen. Das heißt, Programme beschreiben detailliert, *wie* das Ergebnis zu berechnen ist.

Im Gegensatz dazu steht bei der *deskriptiven Programmierung* im Vordergrund zu beschreiben, *was* zu produzieren ist; von der Beschreibung des Berechnungs*weges* wird stärker abstrahiert [63:524]. Zur deskriptiven Programmierung gehören die

[44] ·Deskriptive Programmierung wird oft auch *deklarative Programmierung* genannt.

Varianten *logische Programmierung* und *funktionale Programmierung*, mit Prolog beziehungsweise Lisp und Haskell als bekannten Sprachen.

Bei der *funktionalen Programmierung* werden Programme durch eine Menge von Funktionen definiert, die sich eng am mathematischen Funktionsbegriff orientieren, das heißt eine Funktion ist eine rechtseindeutige Abbildung einer Definitionsmenge in eine Wertemenge [76:15].[45] Funktionen bestehen aus Termen, die Konstanten, Variablen und Funktionsanwendungen enthalten können [76:20f.]. Wie von der Mathematik gewohnt, hängt der Wert eines Ausdrucks ausschließlich von den Werten der verwendeten Variablen und Konstanten ab, nicht jedoch davon, welche Ausdrücke anderweitig ausgewertet wurden oder werden. Bei gegebenen Parametern ergibt die Auswertung eines solchen Terms immer den gleichen Wert, das heißt der Term stellt diesen Wert dar. Terme können somit in einem gegebenen Kontext von Variablen- und Konstantenwerten ohne weitere Auswirkungen durch ihre Werte ersetzt werden [10:3]. Dies nennt man *Bezugstransparenz* (engl. *referential transparency*) [63:524]. Die Anweisungen der imperativen (zum Beispiel der prozeduralen) Programmierung können im Unterschied zu den Termen der funktionalen Programmierung zusätzlich zur Berechnung von Werten weitere Auswirkungen haben, die dazu führen, dass sie trotz gleicher Parameter nicht immer zum gleichen Ergebnis kommen. Diese *Nebenwirkungen* (engl. *side effects*) verändern beispielsweise den Wert von Variablen, die in der Anweisung verwendet werden, und damit auch das Ergebnis der Anweisung.

Funktionen werden bei der funktionalen Programmierung wie gewöhnliche Datenstrukturen behandelt, das heißt sie können wie diese erzeugt, als Parameter übergeben, als Ergebnis zurückgeliefert und in Datenstrukturen gespeichert werden [63:524]. Man spricht auch davon, dass Funktionen „Objekte ersten Ranges“ (engl. *first-class objects*) sind, weil die Möglichkeiten ihrer Verarbeitung gegenüber Datenobjekten nicht eingeschränkt sind.[46] Funktionen, die Funktionen als Parameter verwenden oder Funktionen als Wert liefern, werden *Funktionen höherer Ordnung* genannt [63:524f.].

[45] Pepper [76:15] spricht fälschlich von „linkseindeutig“, meint aber offensichtlich „rechtseindeutig“ [92:2].

[46] Objekte sind hier im allgemeinen Sinn, als „Dinge“, gemeint, nicht im speziellen Sinn der objektorientierten Programmierung.

2.4.1 Funktionale Programmiersprachen

In Bezug auf die Frage, was ·funktionale Programmierung als Paradigma ausmacht, stellt Pepper [76:2] fest: „Leider sind aber bisher alle Versuche, so etwas wie eine formal-mathematische Definition für den Unterschied zwischen funktional und imperativ zu geben, gescheitert.“ Dementsprechend ist es teilweise umstritten, ob eine bestimmte Sprache der funktionalen Programmierung zugerechnet werden kann. Zudem kann eine Allzwecksprache nicht ausschließlich mit mathematischen Funktionen auskommen. Insbesondere die Interaktion mit der Außenwelt durch Ein- und Ausgabe von Daten lässt sich nicht durch reine Funktionen darstellen.[47] Auf die Frage „***Sind wir also an die Grenzen unseres Paradigmas gestoßen?***“ antwortet Pepper [76:243]: „Sicherlich dann, wenn wir puristisch an einem Dogma der Bauart kleben: ‚Funktionales Programmieren heißt, mit dem mathematischen Funktionsbegriff arbeiten.‘ Wenn wir die Sache aber pragmatischer sehen, dann steht nicht mathematische Prinzipienreiterei im Vordergrund, sondern die Hoffnung, durch eine Orientierung an mathematischen Konzepten größere Eleganz, Klarheit und Korrektheit zu gewinnen.“

In dieser Arbeit gelten daher als *funktionale Programmiersprachen* solche, die den funktionalen Ansatz der Programmierung ***unterstützen***. Konkrete Sprachen sind meist nicht einem einzelnen Programmierstil zuzuordnen, sondern vereinen Elemente verschiedener Ansätze [12:7] [58:1] (beispielhaft dargestellt durch die graue Fläche in Abb. 2.10 auf der nächsten Seite). Zudem wurden einige Merkmale oder Sprachkonstrukte aus dem Bereich der funktionalen Programmierung in Sprachen übernommen, die ihren Ursprung in der imperativen Programmierung haben (zum Beispiel „anonyme Funktionen“ in C++11), und neuere Sprachen wurden als Mischformen mehrerer Ansätze, auch des funktionalen, entwickelt (zum Beispiel Ruby oder C♯).

2.4.1.1 Die Sprache Erlang

Das in dieser Arbeit untersuchte Softwaresystem ist in der funktionalen, dynamisch getypten und strikt ausgewerteten Sprache Erlang geschrieben [58:240f.]. Erlang wird seit 1987 von der Firma Ericsson für Anwendungen im Telekommunikationsbereich entwickelt, die einen unkomplizierten und effizienten Umgang mit

[47] Siehe Pepper [76:243ff.] zu verschiedenen Lösungsansätzen.

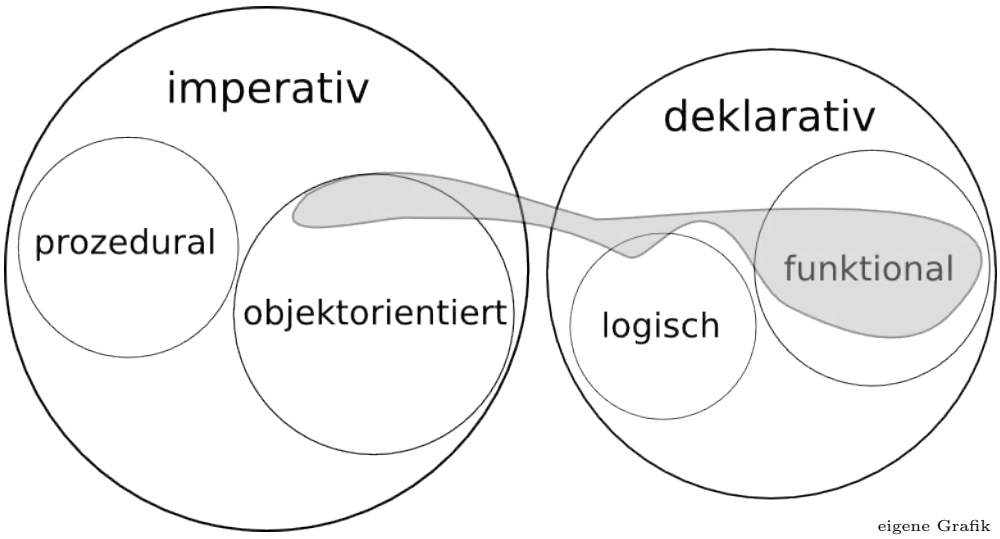


Abbildung 2.10 – Ungefähre Einteilung von Programmierparadigmen und -sprachen. Die Kreise stellen Programmierparadigmen dar, wobei die Größen ungefähr der Verbreitung nach geordnet sind. Die graue Fläche deutet eine konkrete Programmiersprache an.

nebenläufigen Prozessen erfordern [58:240f.]. ERLANG wurde unter anderem durch die logische Programmiersprache PROLOG beeinflusst, der es syntaktisch ähnelt. Die Sprache entspricht in folgenden Aspekten der funktionalen Programmierung:

- Variablen haben in ihrem Gültigkeitsbereich entweder genau einen festen Wert, oder gar keinen [58:241]. Das bedeutet, dass in Bezug auf Variablen ·Bezugstransparenz gilt, was insbesondere den Umgang mit Nebenläufigkeit vereinfacht.
- Funktionen sind „Objekte ersten Ranges“, das heißt sie können als Variablen, als Parameter und als Ergebnis von Funktionen auftreten.
- Jeder Ausdruck repräsentiert einen Wert. (Allerdings haben Ausdrücke nicht unbedingt immer den gleichen Wert.)

Ein ERLANG-Programm besteht aus *Modulen*, in denen *Funktionen* definiert werden. Funktionen müssen *exportiert* werden, um von anderen Modulen aus zugreifbar zu sein. Funktionenen anderer Module können *importiert* werden, um wie lokale verwendet zu werden. Wie bei anderen Sprachen können gemeinsam genutzte Definitionen in sogenannte *Header*-Dateien ausgelagert werden.

Eine Funktion besteht aus einer oder mehreren *Klauseln*. Listing 2.1 auf der nächsten Seite zeigt ein Modul namens `mod_bsp` mit einer Funktion, die aus zwei Klau-

Skript 2.1 – Ein ERLANG-Modul mit Funktionen `fakul/1` und `doppler/0`

```
-module(mod_bsp).

fakul(X) when X < 0 -> throw(error);
fakul(0) -> 0;
fakul(N) -> N * fakul(N - 1).

doppler() -> fun(X) -> 2 * X end.
```

seln besteht, und einer, die aus einer einzigen Klausel besteht. Klauseln bestehen aus Kopf und Rumpf, die durch `->` getrennt sind, im Beispiel etwa `fakul(N)` und `N * fakul(N - 1)`. Der Kopf enthält den Namen der Funktion und die Parameterliste, der Rumpf ist ein Ausdruck, der den Wert der Klausel definiert. Funktionen werden systemweit durch ihr Modul, ihren Namen und ihre Stelligkeit eindeutig bezeichnet, mit folgender Schreibweise: `mod_bsp:fakul/1`. *Anonyme Funktionen* können durch `fun`-Ausdrücke wie `fun(X) -> 2 * X end` dynamisch während der Ausführung des Programms erzeugt, Variablen zugewiesen oder als Wert von Funktionen zurückgegeben werden. Die Funktion `doppler/0` im Beispiel liefert als Wert jeweils eine neue Funktion, deren Wert das Doppelte ihres Argument ist.

Fallunterscheidungen werden durch `case ... of ...` oder `if ... then ...` definiert, Ausnahmebehandlung kann mittels `try ... catch ...` erfolgen. Durch `spawn(X)` wird die der Variablen X zugewiesene Funktion als neuer Prozess gestartet. Wird mit `Pid = spawn(X)` die Prozessnummer dieses Prozesses in der Variablen `Pid` gespeichert, können diesem Prozess mit dem *Sende-Operator* `!` Nachrichten gesandt werden: `Pid ! 42` würde die Zahl 42 als Nachricht versenden. Der Prozess kann mittels `receive` auf bestimmte Nachrichten warten und reagieren:

Skript 2.2 – Senden und Empfangen von Nachrichten in ERLANG

```
self() ! hello
receive
        stop -> halteAn();
        42 -> wasIst(42);
        X -> tuWas(X)
end
```

Im gekünstelten Beispiel schickt der Prozess zunächst eine Nachricht an sich selbst

(`self()`), empfängt dann diese Nachricht und reagiert in Abhängigkeit davon, was er empfangen hat.

2.4.2 Besonderheiten in Bezug auf Softwaremaße

Bezugstransparenz führt dazu, dass es bei funktionaler Programmierung keine Schleifen mit Laufvariablen geben kann, da die Laufvariable ihren Wert nicht ändern könnte. Dies führt dazu, dass anstelle von Schleifenkonstrukten rekursive Funktionen verwendet werden – vermutlich deutlich häufiger als bei imperativer Programmierung. Zudem können Funktionen höherer Ordnung definiert werden, die Funktionen als Parameter verwenden und auf andere Parameter, zum Beispiel Listen, anwenden.

In imperativen Programmen werden zuweilen nacheinander verschiedene Anweisungen ausgeführt, die das gleiche Datenobjekt schrittweise verändern, zum Beispiel Bilddaten nacheinander verschiedenen Transformationen unterziehen. Auch dies ist bei der funktionalen Programmierung durch die Bezugstransparenz ausgeschlossen und muss anders gelöst werden, zum Beispiel indem mit jedem Schritt ein neues Datenobjekt erzeugt wird, das vom nachfolgenden Schritt nicht verändert, sondern nur dazu verwendet wird, ein transformiertes Datenobjekt zu erzeugen.[48]

Diese und weitere Unterschiede in den verfügbaren Ausdrucksmitteln führen dazu, dass funktionale und imperative Programme, auch wenn sie die gleiche Aufgabe lösen, sich stark unterscheiden können. Daher ist es notwendig, Softwaremaße, die in Bezug auf imperative Programme validiert wurden, auch in Bezug auf funktionale Programme zu validieren, um Schlußfolgerungen aus ihnen ziehen zu können.

[48] Es geht hier um das konzeptuelle Vorgehen. Wenn ein neues ***logisches*** Objekt erzeugt wird, muss der betroffene Speicherinhalt nicht unbedingt tatsächlich neu geschrieben werden. Das konzeptuelle Vorgehen effizient umzusetzen, ist Aufgabe des Laufzeitsystems der Programmiersprache.

3 Methoden und Werkzeuge

Nachdem in Kapitel 1, S. 6 die vorliegende Arbeit motiviert und eingeordnet wurde und in Kapitel 2, S. 15 Grundbegriffe der Software-Qualität und der Softwaremessung sowie Methoden der Modellierung von Softwareprodukten und Vorgehensweisen zur Validierung von Softwaremaßen vorgestellt und einige Besonderheiten der funktionalen Programmierung bezüglich der Softwaremessung diskutiert wurden, werden nun die untersuchten Maße interner und externer Qualitätsmerkmale definiert und das verwendete Analysewerkzeug RefactorErl präsentiert.

3.1 Ausgewählte Maße interner Qualitätsmerkmale

Für die Erfassung interner Qualitätsmerkmale der Software wurde eine Reihe von Maßen ausgewählt, die im Folgenden vorgestellt werden. Da der Schwerpunkt dieser Arbeit nicht auf der Implementierung von Maßen, sondern auf deren Validierung liegt, wurden solche Maße ausgewählt, die weit verbreitet sind und sich mit RefactorErl leicht erfassen lassen.

Es werden nun zunächst die ·Basismaße vorgestellt, gefolgt von darauf aufbauenden ·abgeleiteten Maßen. Zu jedem Maß gehört eine Kurzbezeichnung, die im Rest der Arbeit verwendet wird. Je nachdem, ob das Messobjekt eines Maßes ein ganzes Softwaresystem, ein einzelnes Modul oder eine einzelne Funktion ist, ist die Kurzbezeichnung mit „S“, „M“, „F“ indiziert, bei Anwendbarkeit auf verschiedene Komponenten auch kurz „FM“ oder „FMS“. Die Definitionen sind so angepasst, dass Module und Funktionen die Stelle von prozeduralen und objektorientierten Komponententypen einnehmen: Module stehen für Klassen und Pakete, Funktionen für Prozeduren und Methoden. Hypothesen über Zusammenhänge mit externen Qualitätsmerkmalen werden in Abschnitt 4.3.1, S. 75 aufgestellt.

3.1.1 Basismaße

3.1.1.1 Zeilenanzahl

Das sowohl bekannteste als auch einfachste Maß ist die Anzahl der Zeilen des Programmtexts. Zeilen können unterteilt werden in solche, die ausführbaren Code enthalten, und solche, die nur Kommentare enthalten.

Nichtleere Zeilen LOC_{FMS}[49] ist die Anzahl von Zeilen einer Funktion, eines Moduls oder eines Systems, die nicht leer sind, das heißt ausführbaren Code oder Kommentare enthalten. LOC_{M} eines Moduls ist die Summe der LOC_{F} seiner Funktionen zuzüglich weiterer Zeilen wie Modul- und Exportdeklarationen. LOC_{S} eines Systems ist die Summe der LOC_{M} seiner Module.

RefactorErl betrachtet Zeilen, die nur Leerraum enthalten, nicht als leer. Zudem zählt es Zeilen in Headern nicht mit.

Nichtkommentarzeilen $\text{NCLOC}_{\text{FMS}}$[50] ist die Anzahl der nichtleeren Zeilen, die keine Kommentare sind [28:247].

Kommentarzeilen CLOC_{FMS}[51] ist die Anzahl der Kommentarzeilen [28:247].

3.1.1.2 Anzahl verschiedener Programmelemente

Nach LOC_{FM} sind die einfachsten Maße die, die lediglich die Anzahl bestimmter Programmelemente angeben:

- NumMod_{S} ist die Anzahl der Module in einem System.
- $\text{Included}_{\text{M}}$ ist die Anzahl der in einem Modul eingebundenen Header [79:57].
- $\text{Imported}_{\text{M}}$ ist die Anzahl der Module, aus denen Funktionen in ein Modul importiert wurden [79:57].
- NumMac_{M} ist die Anzahl der in einem Modul definierten Macros [79:57].
- NumRec_{M} ist die Anzahl der in einem Modul definierten Records [79:57].

[49] *Lines of Code*
[50] *Non-Comment Lines of Code*
[51] *Comment Lines of Code*

- NumFun$_{M}$ ist die Anzahl der Funktionen in einem Modul [79:57].[52]
- CallsIn$_{M}$ ist die Anzahl der Aufrufe interner Funktionen eines Moduls aus externen Funktionen, das heißt Funktionen anderer Module [79:58].
- CallsOut$_{M}$ ist die Anzahl der Aufrufe externer Funktionen aus internen Funktionen eines Moduls [79:58].
- Calls$_{M}$ ist die Anzahl der Aufrufe externer und interner Funktionen [79:57].
- NumClauses$_{F}$ ist die Anzahl der Klauseln einer Funktion [79:59].
- FanIn$_{F}$ ist der ·Eingangsgrad einer Funktion [79:60] (siehe Abschnitt 2.2.2, S. 29).
- FanOut$_{F}$ ist der ·Ausgangsgrad einer Funktion [79:60] (siehe Abschnitt 2.2.2, S. 29).
- Returns$_{F}$ ist die Anzahl der Vorgänger des ·Stopknotens im KFG der Funktion, also die Anzahl der Ausdrücke, die den Wert der Funktion potentiell bestimmen [79:60].
- NumAnon$_{F}$ ist die Anzahl ·anonymer Funktionen, die in der Funktion definiert werden [79:60]
- NumSend$_{F}$ ist die Anzahl der ·Sende-Ausdrücke in der Funktion [79:60].
- RecBranch$_{F}$ ist die Anzahl der Stellen, an denen sich eine Funktion ***direkt*** rekursiv aufruft [79:59].

3.1.1.3 Kopplungsbeziehungen

Der Begriff *Kopplung* bezeichnet „die Abhängigkeit von und Interaktion zwischen Modulen“ [78:137]. Kopplung entsteht in Erlang-Programmen durch den Import und den Aufruf von Funktionen zwischen Modulen [vgl. 78:137].

Fortführende Funktions- und Modulkopplungen OutFuns$_{M}$ ist die Anzahl von Funktionen in einem Modul, die Funktionen in anderen Modulen aufrufen. Analog dazu ist OutMods$_{M}$ die Anzahl der Module, deren Funktionen aus diesem Modul aufgerufen werden.[53]

[52] Entgegen der Aussage in [79:57] zählt RefactorErl in der Version 0.9.12.01 dabei alle Funktionen mit, die aufgerufen werden, auch wenn ihre Definition nicht verfügbar ist.
[53] Vergleiche ·Efferent couplings bei Martin [61:262f.].

Hinführende Funktions- und Modulkopplungen InFuns_M ist die Anzahl von Funktionen außerhalb eines Moduls, die Funktionen in diesem Modul aufrufen. InMods_M ist dementsprechend die Anzahl der Module, aus denen Funktionen in diesem Modul aufrufen werden.[54]

3.1.1.4 Länge des längsten nicht-rekursiven Aufrufpfades

MaxCall_F ist definiert als die Länge des längsten nicht-rekursiven Aufrufpfades, der von der Funktion ausgeht [79:58].

3.1.1.5 Verschachtelungstiefe von Kontrollstrukturen

Tiefste Verschachtelung von `case` ·Fallunterscheidungen mit `case` können verschachtelt werden, indem in einem Zweig eine weitere Fallunterscheidung als Wert dieses Zweiges angegeben wird. MaxCase_F ist die maximale Anzahl der Verschachtelungsstufen in einer Funktion [79:58f.].

Tiefste Verschachtelung von `begin/end`, `case`, `fun`, `if` oder `receive` Ebenso wie Fallunterscheidungen können auch Blöcke, anonyme Funktionen, binäre Fallunterscheidungen, Empfangsausdrücke oder Ausnahmebehandlungsstrukturen verschachtelt werden. MaxNest_F ist die maximale Verschachtelungstiefe jeglicher dieser Ausdrücke [79:59].

3.1.1.6 Zyklomatische Komplexität

Für einen gerichteten Graphen $G = (V, E)$ mit p schwachen Zusammenhangskomponenten gibt die *zyklomatische Zahl*

$$v(G) = |E| - |V| + p$$

die Anzahl der linear unabhängigen Zyklen an [73:96].[55]

[54] Vergleiche ·Afferent couplings bei Martin [61:262f.].

[55] Die ·zyklomatische Zahl wird auch als *Zyklenrang* [38:146] oder vereinzelt als *Bettizahl* [37:626] bezeichnet.

McCabe ergänzt den ·Kontrollflussgraphen eines Programms um eine Kante vom Endknoten zum Startknoten. Dadurch wird der KFG stark zusammenhängend und die zyklomatische Zahl dieses Graphen entspricht der Anzahl linear unabhängiger Pfade durch den KFG [62:318]. Diese Zahl wird als *zyklomatische Komplexität* des Programms bezeichnet [62:308] (im Folgenden kurz CYC_F).

Die zyklomatische Zahl gibt auch an, mit wievielen Testfällen eine vollständige Zweigüberdeckung erreicht werden kann [57:239].[56]

3.1.1.7 Rekursivität

Da ERLANG ebenso wie HASKELL kein Schleifenkonstrukt enthält, werden für Iterationen vermutlich häufiger rekursive Funktionen definiert [vgl. 80:135]. Rekursion liegt vor, wenn es einen Zyklus im Aufrufgraphen einer Funktion gibt. Wenn nur die Funktion selbst auf diesem Zyklus liegt, spricht Ryder [80:135] von *trivialer Rekursion*, wenn auf dem Zyklus erst andere Funktionen aufgerufen werden, von *nicht-trivialer Rekursion*.[57]

- IsREC_F gibt an, ob eine Funktion rekursiv ist [80:138] [79:61].
- TRIVREC_F gibt an, ob eine Funktion ·trivial rekursiv ist [80:135]
- $\text{NONTRIVREC}_\text{F}$ gibt an, ob eine Funktion ·nicht-trivial rekursiv ist [80:135]

3.1.2 Abgeleitete Maße

3.1.2.1 Gewichtete Funktionen pro Modul

Für jede Funktion eines Moduls wird mit einem festgelegten Maß μ (zum Beispiel zyklomatische Komplexität, siehe Abschnitt 3.1.1.6, S. 51) ihr Gewicht[58] ermittelt. $\text{WMC}(\mu)_\text{M}$[59] ist die Summe der μ-Gewichte der Funktionen des Moduls.

[56] Balzert [5:482] bezeichnet dies fälschlich als „minimale Anzahl von Testfällen“ – diese hängt aber offenbar vom gewählten Überdeckungsmaß ab. Anweisungsüberdeckung ist auch mit weniger als $v(G)$ Testfällen möglich.
[57] Diese Begriffe sind nicht zu verwechseln mit dem der ·primitiv rekursiven Funktionen [85:109].
[58] Chidamber und Kemerer [16] sprechen von „Komplexität“.
[59] Vergleiche *Weighted Methods per Class* bei Chidamber und Kemerer [16:482].

3.1.2.2 Antwortmächtigkeit

Die *Antwortmenge* sei definiert als die Menge aller Funktionen, die in einem Modul definiert oder direkt aufgerufen werden [vgl. 16:487].[60] RFC_M[61] ist die Mächtigkeit der *Antwortmenge* eines Moduls: $\text{RFC}_\text{M} = \text{NumFun}_\text{M} + \text{CallsOut}_\text{M}$.

3.1.2.3 Modulkopplung

CBO_M[62] ist definiert als die Gesamtzahl der Module, mit denen ein Modul gekoppelt ist [16:486], d.h. die Summe aus OutMods_M und InMods_M abzüglich der Anzahl von Modulen, zu denen sowohl hin- als auch fortführende Kopplungen bestehen.

3.1.2.4 Instabilität

Ein Modul, das von anderen Modulen abhängt, muss jeweils geändert werden, wenn sich deren externe Schnittstelle oder Funktionalität ändert. Je mehr ein Modul von anderen Modulen abhängt, desto häufiger muss es vermutlich geändert werden. Wenn viele Module von einem Modul abhängen, bedeutet das einen Druck, dieses Modul möglichst wenig zu ändern, um Änderungen in den abhängigen Modulen zu vermeiden. Ein Modul, das von keinen anderen Modulen abhängt, von dem aber viele abhängen, heisse daher *stabil*. Ein Modul, das von vielen Modulen abhängt, von dem selbst aber keine Module abhängen, heisse *instabil*.

Auf Grund dieser Überlegungen wird das Instabilitäts-Maß Inst_M als Anteil fortführender Kopplungen an allen Kopplungen eines Moduls definiert (übertragen nach Martin [61:262]. Es können zwei Varianten, für Funktions- und Modulkopplungen, unterschieden werden:

$$\text{InstF}_\text{M} = \frac{\text{OutFuns}_\text{M}}{\text{OutFuns}_\text{M} + \text{InFuns}_\text{M}}$$

$$\text{InstM}_\text{M} = \frac{\text{OutMods}_\text{M}}{\text{OutMods}_\text{M} + \text{InMods}_\text{M}}$$

Das Instabilitätsmaß nimmt Werte von 0, „sehr stabil“, bis 1, „sehr instabil“ an.

[60] Aus Effizienzgründen wird ausdrücklich nicht die transitive Hülle der Aufrufrelation gebildet.
[61] Nach *Response For a Class* bei Chidamber und Kemerer [16:487].
[62] *Coupling between object classes*

3.1.2.5 Operationenstrukturierung

Die durchschnittliche Anzahl von Zeilen pro Funktion,

$$\textsc{AvgLOC}_\text{M} = \frac{\Sigma_\text{F}\textsc{LOC}_\text{F}}{\textsc{NumFun}_\text{M}}$$

soll erfassen, inwiefern der Programmtext auf genügend Funktionen aufgeteilt ist. Sehr hohe Werte deuten vermutlich auf schwer handhabbare Funktionen hin [60:28].

3.1.2.6 Operationenkomplexität

Die durchschnittliche zyklomatische Komplexität pro Zeile,

$$\textsc{AvgCyc}_\text{F} = \frac{\textsc{Cyc}_\text{F}}{\textsc{LOC}_\text{F}}$$

soll erfassen, wieviele Verzweigungen in Funktionen zu erwarten sind [nach 60:28].

3.1.2.7 Kopplungsintensität

Die durchschnittliche Anzahl verschiedener Funktionsaufrufe pro Funktion,

$$\textsc{AvgCalls}_\text{M} = \frac{\textsc{CallsIn}_\text{M} + \textsc{CallsOut}_\text{M}}{\textsc{NumFun}_\text{M}}$$

soll erfassen, wie stark Funktionen miteinander interagieren, also gekoppelt sind. Sehr hohe Werte könnten darauf hinweisen, dass Funktionen nicht mit den richtigen „Partnern“ interagieren [60:29].

3.1.2.8 Kopplungsverteilung

Die durchschnittliche Anzahl aufgerufener Module pro Funktionsaufruf,

$$\textsc{AvgOut}_\text{M} = \frac{\textsc{CallsOut}_\text{M}}{\textsc{CallsIn}_\text{M} + \textsc{CallsOut}_\text{M}}$$

soll erfassen, inwiefern viele Module an der Kopplung beteiligt sind [nach 60:29].

3.2 Ausgewählte Maße externer Qualitätsmerkmale

Für die Erfassung externer Qualitätsmerkmale der Software wurde eine Reihe von Maßen ausgewählt, die im Folgenden vorgestellt werden. Es werden zunächst die ·Basismaße vorgestellt, gefolgt von darauf aufbauenden ·abgeleiteten Maßen. Zu jedem Maß gehört eine KURZBEZEICHNUNG, die im Rest der Arbeit verwendet wird. Je nachdem, ob das Messobjekt eines Maßes ein ganzes Softwaresystem, ein einzelnes Modul, eine einzelne Funktion oder ein einzelner ·Fall ist, ist die Kurzbezeichnung mit „S“, „M“, „F“ oder „I“[63] indiziert, bei Anwendbarkeit auf verschiedene Komponenten kurz „FM“ oder „FMS“.

3.2.1 Basismaße

3.2.1.1 Anzahl von Fallarten

NUMISS(T,V)$_{\text{FMS}}$ ist die Anzahl aller ·Fälle eines ·Typs T, die eine Funktion, ein Modul oder ein System in einer Version V betreffen [nach 36:654].Die Typen der Fälle werden wie folgt abgekürzt: „B“[64] für FEHLER, „N“ für NEUERUNGEN, „T“[65] für AUFGABEN und „I“[66] für VERBESSERUNGEN.

STARTISS(T,V)$_{\text{FMS}}$ ist die Anzahl aller ·Fälle eines ·Typs T, die eine Funktion, ein Modul oder ein System in einer Version V ***erstmals*** betreffen, das heisst die für diese Version eröffnet wurden [nach 59:22].

ENDISS(T,V)$_{\text{FMS}}$ ist die Anzahl aller ·Fälle eines ·Typs T, die eine Funktion, ein Modul oder ein System in einer Version V ***letztmalig*** betreffen, das heisst die während dieser Version geschlossen wurden.

[63] I steht für „Issue“, also Fall.
[64] Wie „Bug“.
[65] Wie „Task“.
[66] Wie „Improvement“ – es wird jeweils aus dem Zusammenhang deutlich, dass kein „Issue“, ·Fall, gemeint ist.

3.2.1.2 Bearbeitungszeit von Fällen

ITT_I[67] ist die Zeitspanne von der Eröffnung bis zur Schließung eines Falls, ein Hinweis auf den Aufwand zur Lösung des zugrundeliegenden Problems [nach 59:22,45].

3.2.2 Abgeleitete Maße

3.2.2.1 Fehleranteil an Fällen

$\textsc{BugRate}(V)_{FMS}$ ist der Anteil von ·Fehlern an allen Fällen, die eine Funktion, ein Modul oder ein System in einer Version betreffen. Wenn keine Fehler oder überhaupt keine Fälle vorliegen, ist $\textsc{BugRate}_{FMS}$ Null.

3.2.2.2 Erstellte Fälle pro Monat

$\textsc{StartRate}(T,V)_{FMS}$ ist die Anzahl neuer Fälle vom Typ T in Version V pro Monat [vgl. 59:22]:

$$\textsc{StartRate}(T,V)_{FMS} = \frac{\textsc{StartIss}(T,V)_{FMS}}{\text{Versions} - \text{Lebenszeit in Monaten}}$$

Zu beachten ist, dass diese Anzahl sowohl vom Fehlergehalt oder Änderungsbedarf der Software als auch von der Nutzungsintensität und anderem abhängt.

3.2.2.3 Projektproduktivität

$\textsc{EndRate}(T,V)_S$ ist die Anzahl gelöster Fälle vom Typ T in Version V pro Monat [nach 8:39]:

$$\textsc{EndRate}(T,V)_{FMS} = \frac{\textsc{EndIss}(T,V)_{FMS}}{\text{Versions} - \text{Lebenszeit in Monaten}}$$

Das Maß ist Null, wenn keine Fälle gelöst werden. Wenn keine zu lösenden Fälle vorliegen, ist das Maß nicht definiert.

[67] "Issue Throughput Time" [59:22,45]

3.2.2.4 Mittlere Fallbearbeitungsdauer

Die mittlere Fallbearbeitungsdauer $\text{MedITT(T,V)}_{\text{FMS}}$ ist der Median der Bearbeitungszeiten aller Fälle vom Typ T, die eine bestimmte Funktion, ein Modul oder ein System betreffen und die während der Lebenszeit einer Version V gelöst wurden. Die Dauer wird in Stunden angegeben.

3.2.2.5 Verbesserungsrate

$\text{ImproveRate(V)}_{\text{FMS}}$ ist der Anteil von ·Verbesserungen an gelösten Fällen in Version V [nach 8:33]. Wenn keine Verbesserungs-Fälle gelöst werden, beträgt die Verbesserungsrate Null. Liegen überhaupt keine zu lösenden Fälle vor, ist das Maß nicht definiert.

3.2.2.6 Backlog management index – BMI

$\text{BMI(V)}_{\text{FMS}}$ ist das Verhältnis gelöster Fälle zu erstellten Fällen pro Monat für Version V [52:106]:[68]

$$\text{BMI(V)}_{\text{FMS}} = \frac{\text{EndIss(*,V)}_{\text{FMS}}}{\max(\text{StartIss(*,V)}_{\text{FMS}}, 1)}$$

Ein BMI-Wert unterhalb von Eins bedeutet, dass sich mehr offene Fälle anhäufen, als Fälle geschlossen werden; bei einem Wert über Eins wird die Menge offener Fälle reduziert.

3.3 RefactorErl als Werkzeug für Abfragen an Programmgraphen

Derzeit existiert nur ein einziges Werkzeug – RefactorErl –, das die Erhebung von Softwaremaßen für die Sprache Erlang umfassend und komfortabel unterstützt. Andere Werkzeuge erfüllen nur Teilaufgaben wie die Erstellung des

[68] Da $\text{StartIss(*,V)}_{\text{FMS}}$ Null sein kann, wird der Nenner hier auf mindestens Eins gesetzt.

·Syntaxbaums (Syntax Tools[69]) oder des ·Aufrufgraphen (xref[70]) und erfordern die Implementierung weiterer Algorithmen, um die erforderlichen Informationen abzufragen und zum Beispiel den ·Kontrollflussgraphen zu erstellen. Die Erstellung von ·Kontrollflussgraphen ist für dynamisch getypte und/oder funktionale Programmiersprachen allgemein deutlich schwieriger als für statisch getypte oder imperative Programmiersprachen [67:1] [88:2]. Daher wurde für diese Arbeit trotz der bei einem Prototypen zu erwartenden Schwierigkeiten das Werkzeug RefactorErl ausgewählt.

RefactorErl[71] dient in erster Linie zur Umstrukturierung von Erlang-Programmen. Aufbauend auf der dafür notwendigen statischen Analyse der Programme erlaubt es auch, Softwaremaße und andere Informationen abzufragen. Es wird seit 2006 am Lehrstuhl für Programmiersprachen und Compiler der Fakultät für Informatik an der Eötvös-Loránd-Universität in Budapest, Ungarn, entwickelt. Für die vorliegende Untersuchung wurde der Prototyp in Version 0.9.12.01 vom 17. Januar 2012 verwendet. Zur Zeit der Fertigstellung dieser Arbeit aktuell ist die Version 0.9.12.04 vom 20. April 2012, deren Änderungen für die Untersuchung irrelevant sind.

Aus dem zu bearbeitenden Programmcode erstellt RefactorErl zunächst einen abstrakten Syntaxbaum. In verschiedenen Analyseschritten wird dieser Baum durch zusätzliche Knoten und Kanten zum sogenannten *Programmgraphen* erweitert, der Ergebnisse der statischen semantischen Analyse enthält: den Aufrufgraphen, statisch analysierbare Eigenschaften dynamischer Konstrukte (zum Beispiel mögliche Datentypen) und Informationen über den Datenfluss [53:269].

3.3.1 Abfragesprache

Zum Abfragen der im Programmgraphen gespeicherten Information dient eine Abfragesprache, die von der internen Struktur des Graphen abstrahiert und eine logische Sicht bietet, die der Struktur von Erlang-Programmen entspricht: Module und Header, die Funktionen, Makros und Records definieren, die wiederum aus Ausdrücken aufgebaut sind. Abb. 3.1, S. 60 stellt diese logische Sicht dar.

[69] `http://www.erlang.org/doc/apps/syntax_tools/`
[70] `http://www.erlang.org/doc/man/xref.html`
[71] `http://plc.inf.elte.hu/erlang/`

Abfragen beschreiben ähnlich wie XPath-Abfragen Schrittfolgen im Programmgraphen, wobei die Schritte mit einem Punkt voneinander getrennt werden: `mods` (oben links in Abb. 3.1 auf der nächsten Seite) wählt alle Modul-Knoten aus, `mods.funs` alle Funktions-Knoten, `mods.funs.vars` alle Variablen-Knoten und so weiter. In jedem Schritt kann die ausgewählte Knotenmenge durch ein Prädikat eingeschränkt werden; beispielsweise wählt `mods[name = mod_bsp].funs[name = fakul]` die Funktion aus Listing 2.1, S. 46 aus. Knoten haben verschiedene Eigenschaften, die (wie `name` im Beispiel) als letzter Schritt abgefragt oder in Prädikaten verwendet werden können (siehe Abb. 3.1 auf der nächsten Seite). Ausführliche Erläuterungen zur Abfragesprache finden sich in [79].

Einige Maße stellt RefactorErl direkt bereit, indem sie als Eigenschaften von Modulen oder Funktionen abgefragt werden können. Leider ist die Abfragesprache für die Definitionen einiger Maße zu eingeschränkt. So bietet sie etwa keine Möglichkeit, beliebige Knoten zu zählen oder in Abfragen Variablen zu verwenden. Abfrageergebnisse können jedoch Erlang-Variablen zugewiesen und dann weiterverarbeitet werden. Für diese Untersuchung wurde daher eine Messumgebung entwickelt, die auf der Grundlage eines relationalen Datenbanksystems kompliziertere Abfragen erlaubt (siehe Anhang B.2, S. 126).

3.3.2 Werkzeugvalidität von RefactorErl

In seiner Rolle als ·Refactoring-Werkzeug wird RefactorErl sowohl vom Entwicklungsteam selbst als auch von anderen Autoren überprüft. Zu ersteren zählen Tejfel u. a. [87], die das Werkzeug spezifikationsbasiertem Testen unterziehen, sowie Bozó u. a. [13], die auf die Überprüfung von Programmtransformationen eingehen. Andere Autoren sind Deckers [20] und Jumpertz [51], die erfolgreiche Ansätze zur Verifizierung von RefactorErl vorstellen.

Derzeit sind keine Programmfehler öffentlich bekannt. Obwohl bisher nicht explizit untersucht, stärkt dies auch das Vertrauen in die Messfunktionalität, da diese auf dem gleichen Programmgraphen beruht, der beim Refactoring verwendet wird. Eine Überprüfung der Messfunktionalität erfolgt in Abschnitt 4.2.4, S. 72.

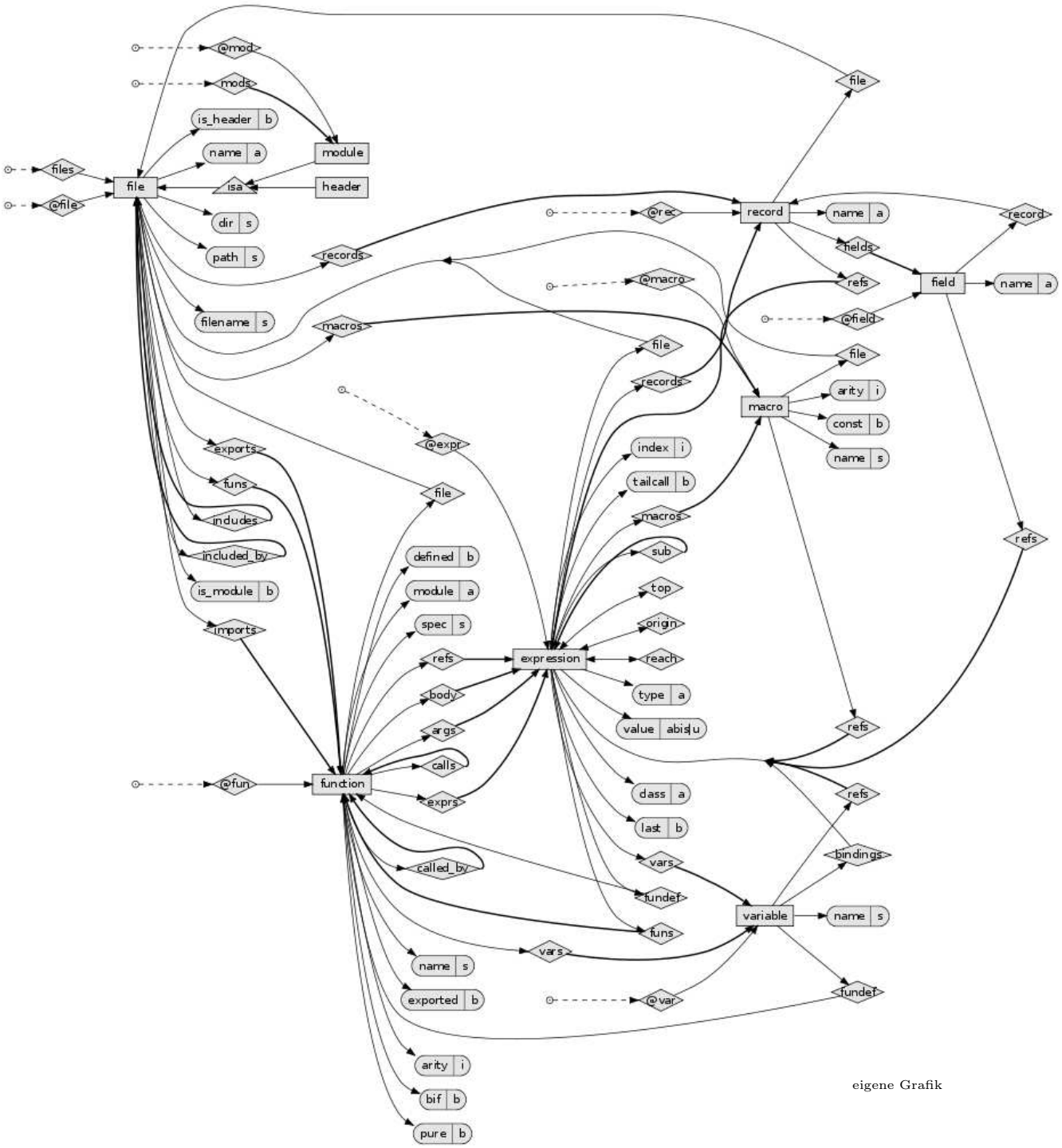


Abbildung 3.1 – Mit RefactorErl abfragbare Relationen und Attribute von Programmkomponenten. Die Darstellung ist an das Entity-Relationship-Model angelehnt. Attribute sind mit ihren Datentypen beschriftet: **a**tom, **b**ool, **i**nt, **s**tring, **u**nknown. Dünne Verbindungslinien bezeichnen 1:1-Beziehungen, fette Linien 1:N-Beziehungen.

4 Empirische Untersuchung

Im vorangehenden Kapitel wurden die zu untersuchenden Maße und das Werkzeug zu ihrer Erhebung behandelt. Nun erfolgt die Vorstellung des untersuchten Softwareprodukts, des Aufbaus der Untersuchung und der Art der gewonnenen Daten. Anschließend werden Hypothesen bezüglich der untersuchten Maße formuliert, die Daten statistisch ausgewertet und auf dieser Basis Aussagen zu den Hypothesen getroffen.

4.1 Untersuchungsobjekt ejabberd

Für die vorliegende Studie muss das Untersuchungsobjekt folgende Eigenschaften haben:

- Der Programmtext und Informationen über bekannte Fehler müssen verfügbar sein.
- Es muss hauptsächlich in Erlang geschrieben sein.
- Es sollte möglichst umfangreich und häufig verwendet sein, um belastbare Ergebnisse zu erzielen.

Große Datenbanken mit fertigen Messergebnissen wie FLOSSMetrics[72] oder PROMISE[73] enthalten keine Erlang-Projekte (und überhaupt kaum Projekte, die funktionale Programmiersprachen verwenden). Mit Ausnahme der Laufzeitumgebung und Standardbibliothek Erlang/OTP selbst sind in Frage kommende Erlang-Projekte deutlich kleiner als Softwaresysteme, die in vergleichbaren Untersuchungen imperativer Programmiersprachen betrachtet werden: einige Zehntausend Zeilen gegenüber mehreren Millionen oder mehr Zeilen.

[72] http://flossmetrics.org/sections/deliverables/WP1
[73] http://promisedata.org/

EJABBERD ist eines der beliebtesten [68:432] Serverprogramme für das EXTENSIBLE MESSAGING AND PRESENCE PROTOCOL, XMPP [82:3] und für seine Skalierbarkeit und Clustering-Fähigkeit bekannt [82:253]. Mit etwa 70000 Zeilen ist es eines der umfangreichsten in ERLANG entwickelten Projekte. Der Programmtext ist für alle 29 Release-Versionen von EJABBERD verfügbar. Zusätzlich enthält das Versionsverwaltungssystem GIT für den Entwicklungszeitraum von über neun Jahren auch alle etwa 2600 Zwischenschritte mit Informationen zu den einzelnen Codeänderungen (*Commits*). Fehleraufzeichnungen liegen im *Fallbearbeitungssystem* (engl. *Issue Tracking System*) JIRA vor. Bis Ende 2010 gab es rund 1500 Einträge in diesem System, die Fehler oder sonstige Änderungsgründe dokumentieren.

EJABBERD wird auch erfolgreich in einem Kommunikationssystem für Tablet-Computer eingesetzt, das seit 2010 bei dem auf mobile Messtechnik und PC-Netzwerktechnik spezialisierten Berliner Unternehmen ESYS GMBH[74] entsteht. Für dieses System hat der Autor eine JAVA-Bibliothek zur Verwendung des XMPP-Protokolls umgesetzt, die den Austausch textueller und grafischer Nachrichten auf der ANDROID-Plattform erlaubt.[75] Die intensive praktische Beschäftigung mit EJABBERD war der Ausgangspunkt für die vorliegende Untersuchung.

4.2 Untersuchungsaufbau

Die empirische Untersuchung besteht aus vier Hauptphasen: Beschaffung der Rohdaten, Analyse und Umformung der Daten, Ermittlung der Messwerte, statistische Analyse der Messwerte. Diese Phasen werden nun vorgestellt. Abb. 4.1 auf der nächsten Seite veranschaulicht den Ablauf von oben nach unten fortschreitend.

4.2.1 Datenbeschaffung

Die Daten der Untersuchung entstammen drei Quellen (in Abb. 4.1 auf der nächsten Seite als Zylinder dargestellt): dem HTTP-Server mit den Release-Version von

[74] `http://www.esys.de`, „Profil“, abgerufen am 26.04.2012.
[75] Die JAVA-Bibliothek und ihre Dokumentation finden sich im elektronischen Anhang dieser Arbeit.

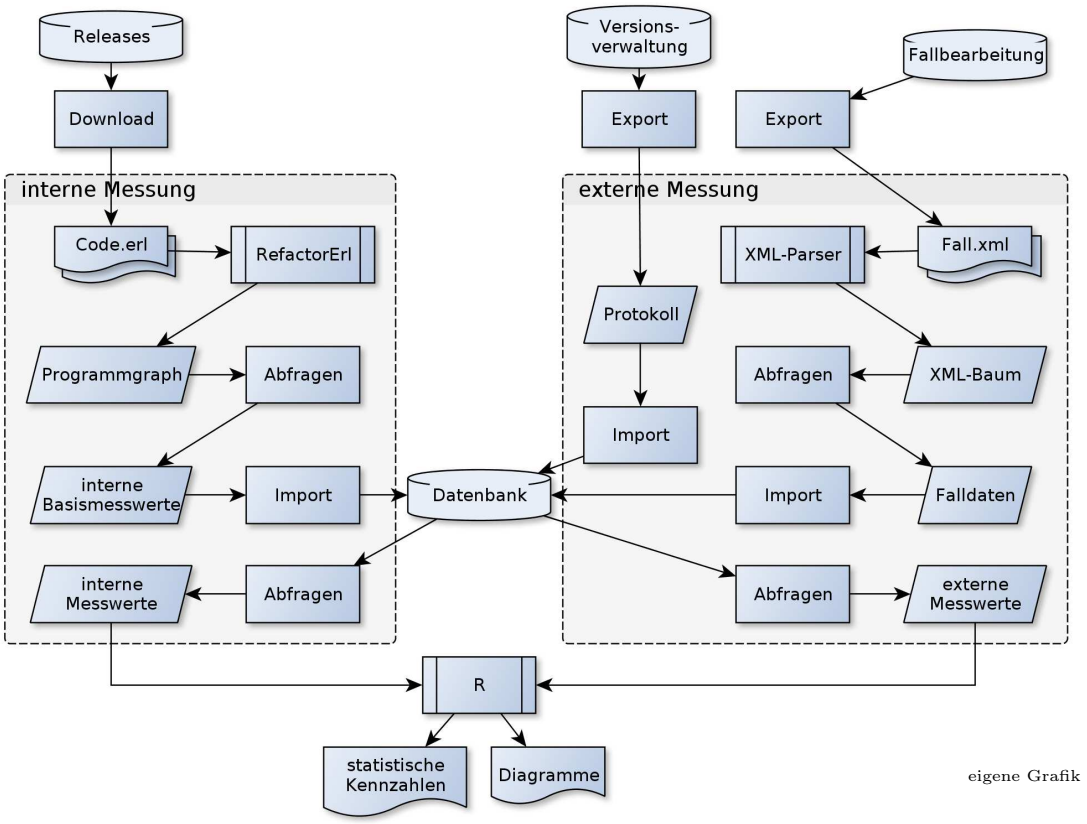


Abbildung 4.1 – Flussdiagramm der empirischen Untersuchung

EJABBERD[76], dem GIT-Versionsverwaltungssystem[77] und dem JIRA-Fallbearbeitungssystem[78].

4.2.1.1 Release-Versionen

Für jede der 29 veröffentlichten Versionen (Releases) von EJABBERD (siehe Tabelle 4.1 auf der nächsten Seite) steht der Programmtext als Paket zum Herunterladen bereit. Der weit überwiegende in ERLANG geschriebene Anteil befindet sich in Modul- und Header-Dateien, hinzu kommen wenige Module, die in ·Abstract Syntax Notation One (ASN.1) verfasst sind und mit dem ERLANG-Compiler zunächst in ERLANG-Code übersetzt werden müssen. Diese ASN.1-Module und alle sonstigen Dateien (unter anderem Skripte und Dokumentation) werden ignoriert.

[76] `http://www.process-one.net/en/ejabberd/archive/`, `http://www.process-one.net/en/ejabberd/downloads`

[77] `git://git.process-one.net/ejabberd/mainline.git`, `https://git.process-one.net/ejabberd/mainline`

[78] `https://support.process-one.net/browse/EJAB`

Tabelle 4.1 – Versionen von EJABBERD, 13.11.2003 bis 24.12.2011

Version	Datum	Nr.	Version	Datum	Nr.	Version	Datum	Nr.
0.5	13.11.03	r.01	1.1.3	07-02-02	r.11	2.1.2	10-01-18	r.21
0.7	04-07-13	r.02	1.1.4	07-09-03	r.12	2.1.3	10-03-12	r.22
0.7.5	04-10-10	r.03	2.0.0	08-02-21	r.13	2.1.4	10-06-04	r.23
0.9	05-04-18	r.04	2.0.1	08-05-20	r.14	2.1.5	10-08-03	r.24
0.9.1	05-05-23	r.05	2.0.2	08-08-28	r.15	2.1.6	10-12-13	r.25
0.9.8	05-08-01	r.06	2.0.3	09-01-15	r.16	2.1.7	11-06-01	r.26
1.0.0	05-12-14	r.07	2.0.4	09-03-13	r.17	2.1.8	11-06-03	r.27
1.1.0	06-04-24	r.08	2.0.5	09-04-03	r.18	2.1.9	11-10-03	r.28
1.1.1	06-04-28	r.09	2.1.0	09-11-13	r.19	2.1.10	11-12-24	r.29
1.1.2	06-09-27	r.10	2.1.1	09-12-17	r.20			

Die Funktionen und Module verschiedener Versionen werden im Folgenden auch kurz „Funktionsversionen“ beziehungsweise „Modulversionen“ genannt.

4.2.1.2 Änderungsprotokoll

Das Versionsverwaltungssystem GIT führt ein Änderungsprotokoll, in dem für jeden ·Commit verschiedene Informationen abrufbar sind, insbesondere der Zeitpunkt der Änderung, eine kurze verbale Beschreibung sowie die einzelnen Änderungen an allen betroffenen Dateien.

An welchen Dateien mit dem Commit welche Änderungen vollzogen wurden, ist bei GIT in Form von PATCH-Dateien abrufbar. In diesen Dateien sind die Nummern der geänderten Zeilen in der ursprünglichen und der geänderten Version des Codes angegeben, sowie die Zeilen selbst mit einem Vermerk, ob sie gelöscht, hinzugefügt oder in sich verändert wurden. Für die empirische Untersuchung werden aus den Dateinamen die Namen der betroffenen Module extrahiert und aus den Zeilenangaben die betroffenen Funktionen ermittelt (siehe Listing B.6, S. 128).

Die verbalen Beschreibungen der Commits verweisen teilweise über die eindeutigen Bezeichner auf den oder die ·Fälle, die durch die Änderungen des Commits bearbeitet werden. Über diese Verweise wird ein Teil der Verknüpfungen von Fällen zu Modulen und Funktionen hergestellt (siehe Abschnitt 4.2.3, S. 69).

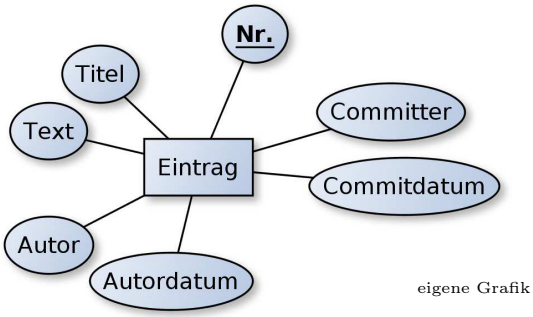


Abbildung 4.2 – ER-Modell eines Log-Eintrags der Versionsverwaltung GIT

4.2.1.3 Fallbearbeitungssystem Jira

Die EJABBERD-Entwickler verwalten anfallende Aufgaben im Fallbearbeitungssystem JIRA. Für die Erhebung der externen Maße (siehe Abschnitt 3.2, S. 55) müssen Informationen aus dem Fallbearbeitungssystem extrahiert werden. Diese sind in verschiedener Form abrufbar, unter anderem als XML-Dateien.[79]

Die Grundstruktur eines Falls wurde bereits in Abschnitt 2.2.4, S. 31 vorgestellt, jedoch ist das Format von Fällen in JIRA flexibel konfigurierbar, so dass zunächst festgestellt werden muss, welche Angaben zu einem Fall in der konkreten JIRA-Konfiguration bei EJABBERD verfügbar sind. Zu diesem Zweck werden die XML-Darstellungen aller Fälle heruntergeladen und über das Werkzeug TRANG[80] eine ·XML-Schema-Definition (XSD) [56:24] generiert. Diese wird manuell in ein Entity-Relationship-Modell eines Falles überführt (siehe Abb. 4.3 auf der nächsten Seite), das alle Eigenschaften eines Falls darstellt. Aus dem Entity-Relationship-Modell wird ein entsprechendes Relationenschema für die Datenbank entwickelt, in der alle Ausgangsdaten und Messwerte hinterlegt sind.

Zur Extraktion von Daten aus Fallbearbeitungssystemen sind dem Autor drei von Dritten entwickelte Werkzeuge bekannt: Luijten [59:8] berichtet in seiner Studie zur Fallbearbeitungseffizienz vom erfolglosen Versuch, das Programm ALITHEIA[81] für den Import von Falldaten einzusetzen. Als Konsequenz daraus entwickelt er ein eigenes System [59:14ff.]. Dieses ist leider nicht öffentlich verfügbar oder über-

[79] Siehe beispielsweise `https://support.process-one.net/si/jira.issueviews:issue-xml/EJAB-1415/EJAB-1415.xml`.
[80] Siehe Anhang B.1, S. 126.
[81] `http://sqo-oss.org`

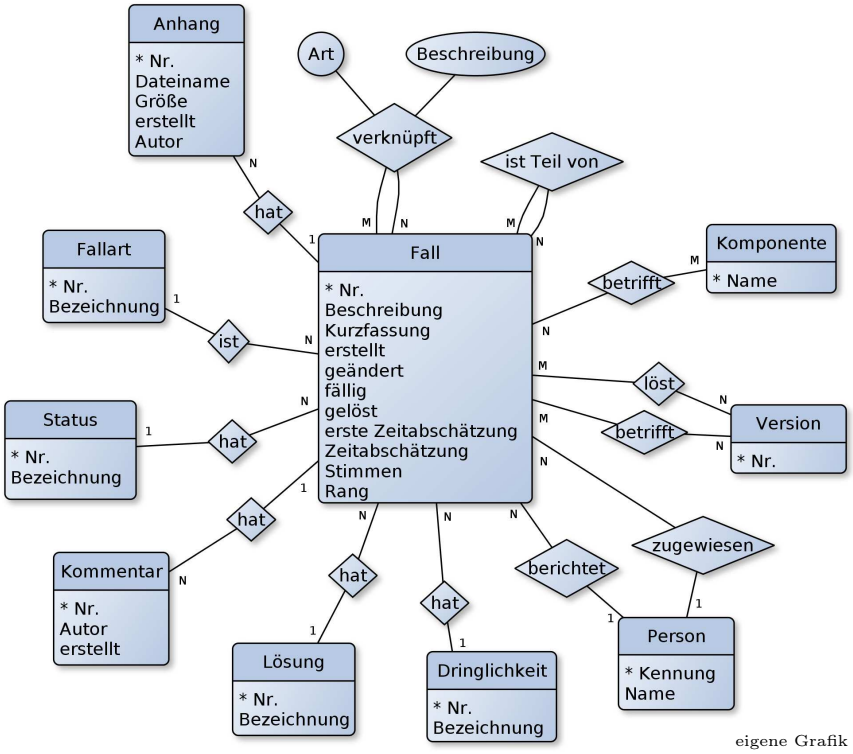


Abbildung 4.3 – ER-Modell eines ·Falls des ·Fallbearbeitungssystems JIRA. (Attribute von Entitäten sind zugunsten der Übersichtlichkeit UML-artig dargestellt. Schlüsselattribute sind mit einem Stern (*) markiert.)

haupt auffindbar. Darüber hinaus verspricht BICHO[82], Fallbearbeitungssysteme zu extrahieren und analysierbar zu machen. Es erwies sich jedoch noch als zu fehlerhaft, um mit vertretbarem Aufwand eingesetzt zu werden, so dass ebenfalls ein eigenes Extraktionsprogramm entwickelt werden musste (siehe Anhang B, S. 126).

Wie in Abschnitt 2.3.4.2, S. 40 erläutert, stellen die Fälle im Fallbearbeitungssystem eine verzerrte Stichprobe aller vorhandenen Probleme oder Änderungsgründe dar, die zudem ***nach*** Veröffentlichung einer Version der Software festgestellt wurden. ***Vor*** Veröffentlichung treten wahrscheinlich deutlich mehr und andersartige Probleme auf. Ryder [80:95] klassifiziert bei seiner Untersuchung alle Codeänderungen manuell als Korrekturen oder Verbesserungen; Bird u. a. [9:129] haben zur Gewinnung verlässlicher Fehlerinformationen einige grundständig Studierte mit der manuellen Klassifizierung der Codeänderungen des APACHE-Projekts beauftragt. EJABBERD ist für die manuelle Klassifizierung durch eine Person zu groß,

[82] https://projects.libresoft.es/projects/bicho

das zweite Verfahren nicht verfügbar. Daher ist im Folgenden ausdrücklich von FEHLERN im Unterschied zu Fehlern die Rede, also von Fällen als Abbild einer Teilmenge der Fehler.

4.2.2 Datenbereinigung

Vor der Analyse der Daten müssen diese in ihrer Form vereinheitlicht und von fehlerhaften Teilen befreit werden.

4.2.2.1 Namensangleichung

In ERLANG sind Modul- und Funktionsbezeichner ·Atome. Da Atome nicht mit Großbuchstaben beginnen dürfen, muss eine großgeschriebene Modul- oder Funktionsbezeichnung in einfache Anführungszeichen eingeschlossen werden, was sie zu einem Atom macht. Da derartige Modulbezeichner nicht mehr mit den Namen der enthaltenden Dateien übereinstimmen, werden für die weitere Untersuchung die einfachen Anführungszeichen entfernt. Betroffen sind genau die zwei Module, die aus der ASN.1-Darstellung erzeugt werden, `'ELDAPv3'` und `'XmppAddr'`, die jedoch ohnehin aus der Untersuchung ausgeschlossen werden, da sie nicht als ERLANG-Code entwickelt, sondern erst vom Compiler in solchen übersetzt werden.

4.2.2.2 Ausschluss unvollständiger oder fehlerhafter Daten

Vor der Auswertung werden unvollständige oder fehlerhafte Daten entfernt. Folgende Kriterien führen zum Ausschluss von Daten:

1. ***Standardmodul:*** Das zugehörige Modul ist Teil der Standardbibliothek, daher kann der Programmtext nicht analysiert werden. Ohnehin müsste jeweils die zeitgenössische Version der Standardbibliothek statt der aktuellen analysiert werden, was über den Rahmen der Untersuchung ginge.
2. ***Standardfunktion:*** Die zugehörige Funktion ist nicht im Programmtext definiert, sondern wird beim Übersetzen automatisch erzeugt. Auch hier kann der Programmtext nicht analysiert werden.

Tabelle 4.2 – Anzahl ausgeschlossener Funktionsversionen mit dem Grund des Ausschlusses

Ausschlussgrund	Anzahl Funktionsversionen
Standardmodul	533
Standardfunktion	1020
Ladefehler	2517
Duplikat	33

3. ***Code konnte nicht geladen werden:*** Die zugehörige Funktionsdefinition konnte von RefactorErl nicht analysiert werden. Gründe dafür sind Syntaxfehler oder die Verwendung nicht definierter Macros. Alle Maße, die eine Analyse des Programmtextes erfordern, sind nicht anwendbar.
4. ***Duplikate:*** Eine geringe Menge von Daten enthielt doppelte Messwerte, die mit einer frühen Version der Messumgebung ermittelt wurden. Diese wurden zugunsten der neueren Messwerte entfernt.

Auf Grund der verschiedenen Ausschlussgründe wurden 4103 Funktionsversionen ausgeschlossen, wie in Tabelle 4.2 aufgeführt.

4.2.2.3 Ausschluss von Version Nr. 26

Die 26. Version von ejabberd, Version 2.1.7, war nur für zwei Tage aktuell, bis sie durch Version 2.1.8 ersetzt wurde, die einen schwerwiegenden Fehler behob.[83] Durch ihre kurze Lebensdauer, in der jedoch 30 Fälle geschlossen wurden, hätte sie alle Maße, die die Veränderungen der Fälle über die Zeit erfassen, verzerrt. Da sie ansonsten identisch mit Version 2.1.8 ist, wird sie aus der Untersuchung ausgeschlossen. Die Alternative, ihre Daten mit denen von Version 2.1.8 zu vereinigen, wurde auf Grund der zu erwartenden Konflikte mehrerer Messwerte für die gleichen Komponenten unterlassen.

Nach Abschluss der Datenbereinigung gehen auf Modulebene 3939 Modulversionen und auf Funktionsebene 66537 Funktionsversionen in die Untersuchung ein.

[83] Siehe `http://www.process-one.net/en/ejabberd/release_notes/release_note_ejabberd_2.1.8/`, zuletzt abgerufen am 9. Mai 2012.

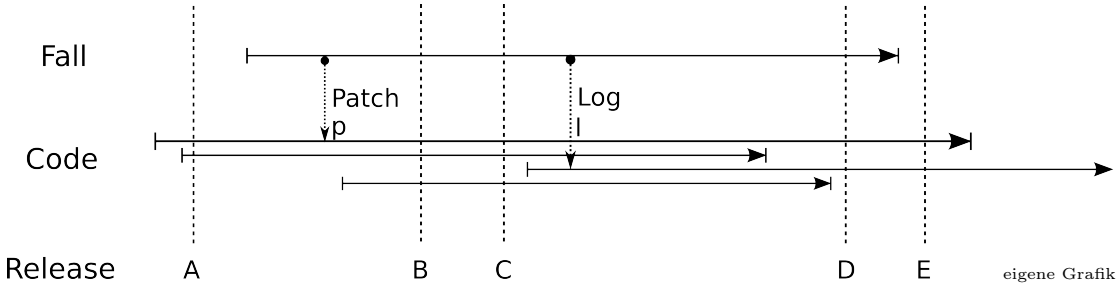


Abbildung 4.4 – Zuordnung eines Falls zu Codeabschnitten über Patches und Änderungsprotokoll. Die Lebensdauer eines Falls oder Codeabschnitts (Modul oder Funktion) wird durch einen Pfeil vom Zeitpunkt der Erstellung bis zur Lösung beziehungsweise Löschung dargestellt. Im Beispiel ist durch einen Patch und einen Eintrag im Änderungsprotokoll bekannt, welche Codeabschnitte ein Fall zu den Zeitpunkten p und l betrifft. Der Fall betrifft diesen Codeabschnitt für alle Releases, deren Lebensdauer sich mit seiner überlappt, das heißt A, B, C und D.

4.2.3 Zuordnung von Fällen zu Modulen und Funktionen

Um Zusammenhänge zwischen internen und externen Messwerten des Softwaresystems untersuchen zu können, muss festgestellt werden, welche internen und externen Messwerte die gleichen Releases, Module oder Funktionen betreffen. Für die internen Messwerte ist ohnehin bekannt, zu welcher Programmkomponente sie gehören; für die Fälle des Fallbearbeitungssystems muss dies erst festgestellt werden.[84] Dies geschieht in zwei Stufen, die nachfolgend erläutert werden:

1. Aus dem Änderungsprotokoll und aus dem Fallbearbeitungssystem werden ***direkte*** Verknüpfungen von Fällen und Programmkomponenten extrahiert.
2. Aus direkten Verknüpfungen und den Gültigkeitszeiträumen von Releases und Fällen werden ***indirekte*** Verknüpfungen abgeleitet.

4.2.3.1 Direkte Verknüpfungen

Im Fallbearbeitungssystem ist lediglich grob angegeben, für welche Versionen und welchen Bereich (etwa „Mehrbenutzerchat“ oder „Dokumentation“) ein Fall relevant ist; daraus lässt sich kaum auf konkrete Module oder Funktionen schließen. Jedoch können Fälle und Programmtext auf zweierlei Arten eindeutig miteinander verknüpft sein: Zum einen durch PATCH-Dateien, die im Fallbearbeitungssystem

[84] Demgegnüber lag diese Information etwa bei der Untersuchung von Graves u. a. [36:655] bereits im Fallbearbeitungssystem vor.

hinterlegt sind, und zum anderen durch Einträge im Änderungsprotokoll, in denen die eindeutige Nummer eines Falls vorkommt.

Die Veröffentlichung eines Patches zu einem bestimmten Zeitpunkt signalisiert, dass die von dem Patch berührten Codeabschnitte Teil des Problems sind, das der Patch lösen soll. Dieses Problem muss zu einem Zeitpunkt vor der Veröffentlichung des Patches entstanden sein. Eine konservative Annahme ist, dass das Problem erst seit der letzten Version vor Veröffentlichung des Patches besteht. Aus der PATCH-Datei werden die betroffenen Dateien und Zeilen ausgelesen. Da in ERLANG Modul- und Dateinamen übereinstimmen, die Module nicht hierarchisch angeordnet sind und die Anfangs- und Endzeile jeder Funktion bekannt ist, kann so eine direkte Verknüpfung eines Falls mit einer Funktion hergestellt werden (siehe Abb. 4.4, S. 69). Das entwickelte Extraktionsprogramm lädt dazu zu jedem Fall die mit ihm als Dateianhang verknüpften PATCH-Dateien herunter und extrahiert Modul- und Funktionennamen. Über das Datum des Anhangs stellt es die seinerzeit gültige EJABBERD-Version fest.

Schließlich trägt es in der zentralen Datenbank eine Verknüpfung des jeweiligen Falles mit den extrahierten Modulen und Funktionen in der entsprechenden Version ein (siehe Abb. 4.4, S. 69).

Im Versionsverwaltungssystem GIT kann zu einem Änderungseintrag ebenfalls eine entsprechende PATCH-Datei erzeugt werden, aus der ebenfalls die betroffenen Module und Funktionen extrahiert werden können. Bei Änderungen, die mehrere Fälle betreffen, können dabei falsche Verknüpfungen entstehen, da in der PATCH-Datei nicht mehr unterschieden werden kann, zu welchem Fall eine Änderung gehört. Luijten [59:16], der Zusammenhänge von Softwaremaßen und Fallinformationen für objektorientierte Programmierprachen untersucht, verzichtet aus diesem Grund auf eine detaillierte Verknüpfung von Fällen zum Programmtext. Bei EJABBERD erwähnen 38 von 2641 Änderungseinträgen mehr als einen Fall; insgesamt sind von diesen Mischverknüpfungen 84 von 1470 Fällen betroffen.[85] Um falsche positive Zuordnungen zu vermeiden, werden die entsprechenden Änderungseinträge bei der Verknüpfung ignoriert.

Mit dieser Technik werden insgesamt 726 der 1470 Fälle, 49 Prozent, mit Programmkomponenten verknüpft. Auf der Funktionenebene werden durch 5840 Ver-

[85] Die Angaben gelten für die Zeit bis zum 17. März 2012.

knüpfungen alle diese Fälle zu 4374 Funktionsversionen zugeordnet, was 6,5 Prozent aller Versionen entspricht. Auf der Modulebene wird durch 3347 Verknüpfungen die gleiche Fallmenge zu 1448 Modulversionen zugeordnet, was 36,7 Prozent entspricht.

Im nächsten Abschnitt wird erläutert, wie dieses Ergebnis noch verbessert wird.

4.2.3.2 Indirekte Verknüpfungen

Unter der Annahme, dass Fälle innerhalb der Lebenszeit der EJABBERD-Version geschlossen werden, während der das zugrundeliegende Problem behoben wurde, gibt das Enddatum des Falls an, bis zu welcher Version der betroffenen Funktionen und Module die Verknüpfung zeitlich nach hinten erweitert werden kann. In Abb. 4.4, S. 69 beispielsweise ist das für den Patch p die Version D, da sie als letzte vor dem Enddatum des mit dem Patch verknüpften Falls veröffentlicht wurde. Auf diese Weise wird die Verknüpfung des Falls, die zunächst direkt über den Patch p mit Version A der betroffenen Funktionen und Module hergestellt wurde, auf alle Versionen bis einschließlich D erweitert. Allgemein formuliert werden indirekte Verknüpfungen eines Falls für alle Versionen erzeugt, deren Lebenszeit mit der des Falls überlappt.

Mit diesem Verfahren, dass in der Literatur bisher nicht vorgefunden wurde, verdreifacht sich die Anzahl der verknüpften Funktionsversionen auf 14571, während die verknüpfte Modulmenge auf das 1,4-Fache, 2015 Modulversionen, wächst. Durch direkte und indirekte Verknüpfungen wurden somit 21,8 Prozent aller Funktionen und 51,2 Prozent aller Module verknüpft. Die Anzahl der verknüpften Fälle blieb offensichtlich unverändert. Tabelle 4.3 auf der nächsten Seite zeigt die Verteilung aller ·Fälle und der verknüpften Fälle auf die verschiedenen Typen (siehe Abschnitt 2.2.4, S. 31), sowie den jeweiligen Anteil an allen Verknüpfungen, sowohl auf Funktions- als auch auf Modulebene. Die Übereinstimmung der Verteilungen der verknüpften Fälle mit denen aller Fälle deutet darauf hin, dass die Stichprobe der verknüpften Fälle nicht verzerrt ist. Auffällig ist, dass NEUERUNGEN und VERBESSERUNGEN überproportional viele Verknüpfungen haben, also mit überproportional vielen Programmkomponenten in Bezug stehen. Diese Beobachtung könnte bei weiteren Untersuchungen vertieft werden (siehe Abschnitt 7.2, S. 110).

Tabelle 4.3 – Verteilung der Fälle und Verknüpfungen auf die verschiedenen Typen (Angaben in Prozent. Summen von mehr als 100 Prozent auf Grund von Rundung.)

Typ	Anteil an Fällen: alle	verknüpfte	Anteil an Verknüpfungen
FEHLER	52	49	22
NEUERUNG	14	14	29
AUFGABE	7	3	4
VERBESSERUNG	30	34	45

4.2.4 Werkzeugvalidität des Messsystems

Bei einem Prototypen wie REFACTORERL sind Fehler noch weniger ausgeschlossen als bei ausgereifteren Werkzeugen. Während REFACTORERL als Refactoring-Werkzeug wiederholt überprüft wird (Abschnitt 3.3, S. 57), sind bisher keine Untersuchungen zur gezielten Überprüfung der Messfunktionalität bekannt. Für die Untersuchung der ·Werkzeugvalidität gilt hier als Werkzeug das gesamte Messsystem aus REFACTORERL und sämtlichen Hilfsprogramme und -skripte. Nicht näher betrachtet werden die verwendeten Standardprogramme und -bibliotheken (Anhang B.1, S. 126), deren korrekte Funktion vorausgesetzt wird.

Um die Messergebnisse stichprobenartig zu überprüfen, werden zu allen 29 Release-Versionen von EJABBERD folgende Werte unabhängig von REFACTORERL ermittelt:

1. Anzahl der Module
2. Anzahl der Funktionen
3. Anzahl nicht-leerer Zeilen

Diese Vergleichswerte wurden gewählt, weil sie sich leicht ermitteln lassen, so dass Fehler im Kontrollwerkzeug ausgeschlossen werden können. Allerdings sind die Werte offensichtlich nicht voneinander unabhängig: Module, die nicht korrekt importiert wurden, führen in allen drei Vergleichen zu Abweichungen; nicht korrekt importierte Funktionen führen auch zu Abweichungen bei der Zeilenanzahl.

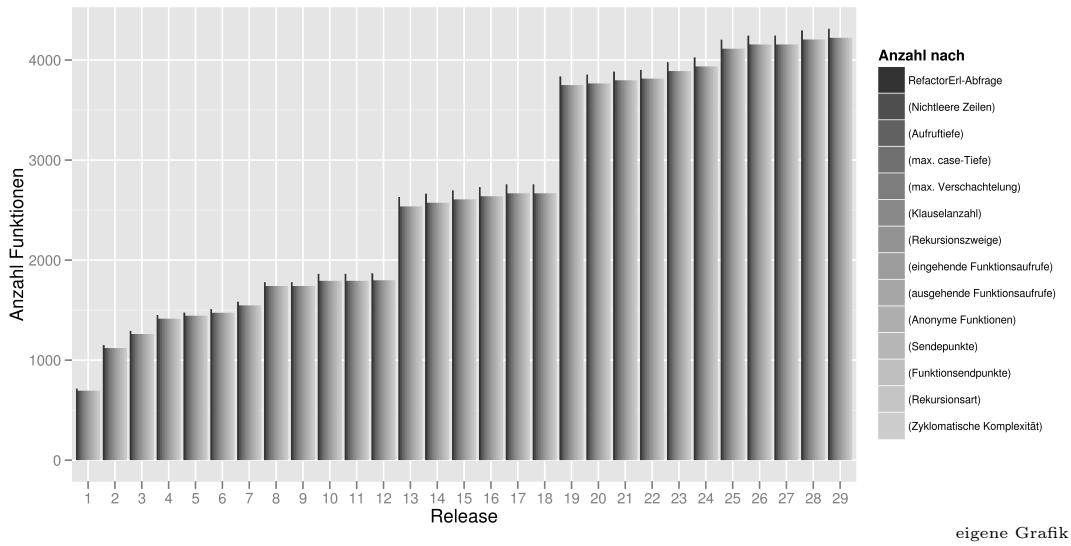
Abweichungen können entstehen, weil REFACTORERL nur Code importieren kann, der in der Betriebsumgebung auch kompilierbar ist. Wenn beispielsweise Macros aus fehlenden Bibliotheken verwendet werden, schlägt der Import fehl, so dass

für das betroffene Modul oder seine Funktionen keine oder nur eingeschränkte Informationen abgefragt werden können.

4.2.4.1 Anzahl der Module pro Version

Für jedes Release wird die Anzahl der durch RefactorErl importierten Module mit der Anzahl der Erlang-Dateien (siehe Listing B.1, S. 127) verglichen. Dies ist eine gute Näherung, da jedes Modul in einer separaten Datei liegen muss und jede Erlang-Datei ein Modul enthalten muss. Die Ergebnisse beider Zählungen stimmen über, womit der erste Test der Werkzeugvalidität erfolgreich ist.

4.2.4.2 Anzahl der Funktionen pro Version



eigene Grafik

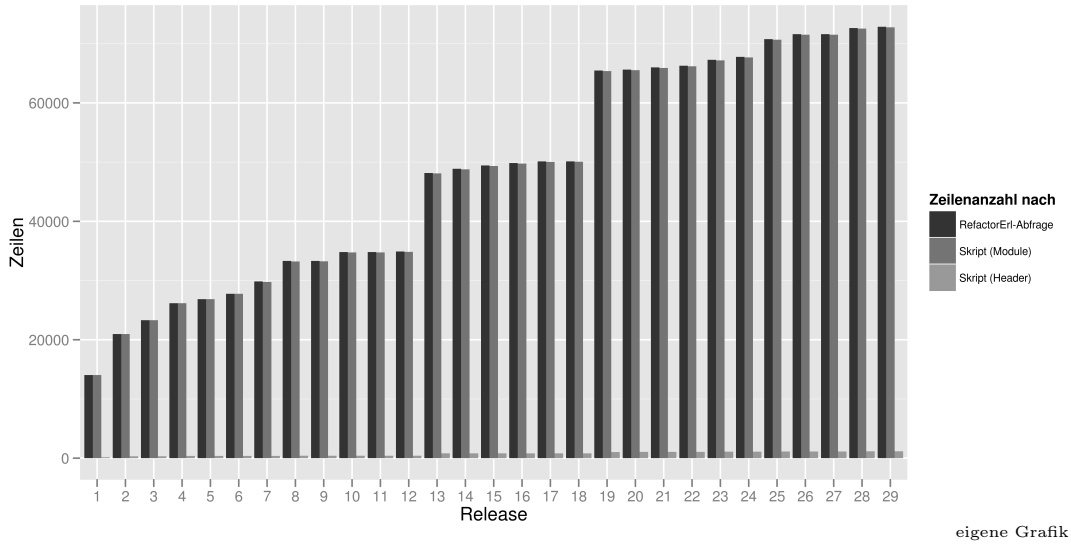
Abbildung 4.5 – Vergleich der Anzahl der Funktionen in ejabberd. Die Anzahl der Funktionen wurde einerseits mit RefactorErl abgefragt (schwarze Säulen), andererseits wurden die Funktionen gezählt, für die Messwerte der Standardmaße von RefactorErl vorliegen (graue Säulen, vgl. auch Listing B.3, S. 128).

Die Anzahl der Funktionen wird für jedes Release mittels RefactorErl am Programmgraphen abgefragt und mit der Anzahl der importierten Funktionen (Listing B.3, S. 128) verglichen. Dabei werden jeweils nur Funktionen aus solchen Modulen gezählt, die ohne Fehlermeldung importiert wurden. Abb. 4.5 zeigt, dass die Anzahl der Funktions-Messwerte für jedes Maß identisch ist. Die Messwerte von

RefactorErl sind durchgängig geringfügig höher, weil es auch nicht-definierte aufgerufene Funktionen mitzählt.

4.2.4.3 Anzahl nicht-leerer Zeilen pro Version

Für die Zeilenanzahl LOC in jedem Release werden die Angaben von RefactorErl mit den Ergebnissen von Listing B.5, S. 128 verglichen. Die Bestimmung der Zeilenanzahl ist für RefactorErl nicht trivial, da es den ursprünglichen Programmtext zum Zählen der Zeilen erst wieder aus dem Programmgraphen rekonstruieren muss. Daher ist dies ein akzeptabler Test für die Korrektheit des Programmgraphen, an dem alle Basismaße abgefragt werden.

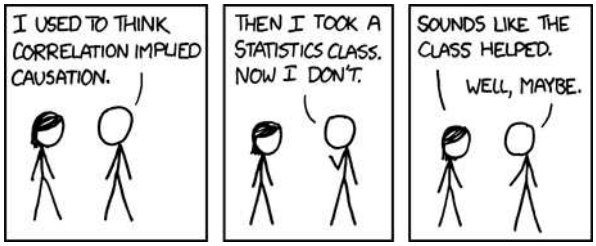


eigene Grafik

Abbildung 4.6 – Vergleich der Zeilenanzahl LOC in ejabberd, ermittelt durch RefactorErl und Listing B.5, S. 128.

Die Ergebnisse von Listing B.5, S. 128 stimmen weitgehend mit denen der Messumgebung überein (siehe Abb. 4.6). Die Differenzen betragen vernachlässigbare zwei Zeilen in einem einzigen Modul ab Release Nr. 7, wofür allerdings keine Erklärung gefunden werden konnte.

4.3 Statistische Untersuchung



Grafik: [71:110120]

Im Folgenden werden alle Funktionen und alle Module aus den verbliebenen 28 Versionen[86] von EJABBERD gemeinsam betrachtet. Es wird davon abgesehen, die einzelnen Versionen getrennt zu betrachten, um den unbekannten Einfluss der wechselnden Intensität der Entwicklungstätigkeit und der Benutzung auszugleichen. Dieses Vorgehen entspricht dem von Hopkins und Hatton [44]. Ryder [80] hingegen reduziert die Daten der gesamten Lebenszeit der von ihm untersuchten Systeme auf die Maximalwerte der internen Maße pro Funktion und auf die Gesamtanzahl von Korrekturen bis zu einem festen Zeitpunkt (siehe Abschnitt 1.2.0.2, S. 10). Auf diese Weise sind niedrige Messwerte und Korrekturanzahlen in seiner Datenmenge unterrepräsentiert. Demgegenüber hat der hier gewählte Ansatz den Vorteil, alle Zwischenstadien der Entwicklung der Programmkomponenten zu berücksichtigen. Je nach betrachtetenen Maßen wird allen Modulen oder allen Funktionen ihr interner Messwert zugeordnet, sowie über die mit ihnen verknüpften Fälle ihr externer Messwert.

4.3.1 Hypothesen

Die untersuchten Hypothesen entsprechen den Validierungskriterien ·Assoziation, ·Trennschärfe und ·Parallele Veränderung (siehe Abschnitt 2.3.2.2, S. 36). Sie sind entweder direkt von anderen Autoren übernommen (insbesondere von Ryder [80] als der einzigen bekannten vergleichbaren Arbeit mit Bezug auf funktionale Programmierung) oder aus informellen Programmierrichtlinien abgeleitet (*Program Development Using Erlang - Programming Rules and Conventions* [26] und Cesarini und Thompson [15]). Die Hypothesen betreffen nur die Funktions- oder Modulebene. Ganze Systeme werden nicht betrachtet.

[86] Siehe Abschnitt 4.2.2.3, S. 68.

Das ·gewählte Signifikanzniveau ist $\alpha = 0,05$. In Anbetracht der Ergebnisse von Berg [6] und Ryder [80] (siehe Abschnitt 1.2, S. 7) werden Korrelationskoeffizienten mit einem Betrag unter $0,2$ als vernachlässigbar angesehen, solche mit einem Betrag zwischen $0,2$ und $0,5$ als „schwach“ bis „moderat“ bezeichnet und alle betragsmäßig größeren als „stark“.

Falls nicht anders vermerkt, gilt für das Kriterium ·Assoziation als Nullhypothese, dass die Werte des Korrelationskoeffizienten betragsmäßig kleiner oder gleich $0,2$ sind. Bezüglich des Kriteriums ·Assoziation sind die vorgestellten Forschungshypothesen als Alternative zu dieser Nullhypothese zu verstehen. Wenn die hypothetische Korrelation nicht näher bezeichnet wird, ist also betragsmäßig eine Stärke $> 0,2$ gemeint.

Für das Kriterium ·Parallele Veränderung wird in einem ersten Schritt gegen die Nullhypothese getestet, dass der Rangkorrelationskoeffizient gleich Null ist. Im zweiten Schritt wird die Stärke der Rangkorrelation nach obiger Regel bewertet. Das Kriterium gilt als erfüllt, wenn eine signifikante Korrelation beobachtet wird, deren Stärke $0,2$ übersteigt.

Für das Kriterium ·Trennschärfe genügt der Nachweis eines signifikanten Zusammenhangs; gegebenenfalls wird zusätzlich die Stärke des Zusammenhangs eingeordnet.

4.3.1.1 Grundannahmen

Informelle Aussagen zu externen Qualitätseigenschaften in der verwendeten Literatur werden nach folgende Überlegungen, die in dieser Arbeit nicht überprüft werden können, operationalisiert:

1. Wenn die externe Eigenschaft nicht genauer angegeben ist, wird davon ausgegangen, dass ·Wartbarkeit oder einer ihrer Aspekte gemeint ist. „Verständlichkeit“ wird als einer dieser Aspekte angesehen.
2. Für folgende externe Maße wird ein positiver Zusammenhang mit Wartbarkeit unterstellt:
   - EndRate(T,V)$_{\text{FM}}$ – die durchschnittliche Anzahl gelöster Fehler pro Monat.

- ImproveRate(V)$_{FM}$ – der Anteil von Verbesserungen an gelösten Fällen während der Lebenszeit einer Version.
- BMI(V)$_{FM}$ – der ·Backlog Management Index, also das Verhältnis gelöster Fälle zu neu eröffneten Fällen.

3. Ein negativer Zusammenhang mit Wartbarkeit wird für folgende externe Maße angenommen:
   - NumIss(B)$_{FM}$ – die Anzahl an Fällen für eine Version einer Funktion oder eines Moduls.
   - BugRate$_{FM}$ – der Anteil der Fehler an allen Fällen einer Version einer Funktion oder eines Moduls.
   - MedITT(B)$_{FM}$ – die mittlere Bearbeitungszeit von Fehlern bei einer Version einer Funktion oder eines Moduls.

Die nachfolgend angegebenen Quellen beziehen sich auf die Aussagen, von denen die Hypothesen abgeleitet wurden.

4.3.1.2 Hypothesen über Assoziation und parallele Veränderung

Um die Validierungskriterien ·Assoziation und ·Parallele Veränderung zu überprüfen, wird die Korrelation und Rangkorrelation von Paaren von Messwerten untersucht, wobei eines der Maße intern und das andere extern ist.

Hypothese AP1 Chidamber und Kemerer [16:482] mutmaßen sinngemäß, dass WMC(Cyc)$_M$, die Summe der zyklomatischen Komplexität aller Funktionen eines Moduls, positiv mit ·Wartbarkeit zusammenhängt. Daher werden folgende Unterhypothesen aufgestellt:

1. WMC(Cyc)$_M$ korreliert positiv mit NumIss(B)$_M$.
2. WMC(Cyc)$_M$ korreliert positiv mit BugRate$_M$.
3. WMC(Cyc)$_M$ korreliert positiv mit MedITT(B)$_M$.
4. WMC(Cyc)$_M$ korreliert negativ mit BMI$_M$.
5. WMC(Cyc)$_M$ korreliert negativ mit EndRate(B)$_M$.
6. WMC(Cyc)$_M$ korreliert negativ mit ImproveRate$_M$.

Hypothese AP2 Auch für CBO_M vermuten Chidamber und Kemerer [16:486] einen positiven Zusammenhang mit der ·Wartbarkeit von Software [vgl. 5:474]. Daraus ergeben sich analog zu $\text{WMC}(\text{CYC})_\text{M}$ folgende Unterhypothesen:

1. CBO_M korreliert positiv mit $\text{NUMISS}(\text{B})_\text{M}$.
2. CBO_M korreliert positiv mit BUGRATE_M.
3. CBO_M korreliert positiv mit $\text{MEDITT}(\text{B})_\text{M}$.
4. CBO_M korreliert negativ mit BMI_M.
5. CBO_M korreliert negativ mit $\text{ENDRATE}(\text{B})_\text{M}$.
6. CBO_M korreliert negativ mit $\text{IMPROVERATE}_\text{M}$.

Hypothese AP3 Chidamber und Kemerer [16:487] argumentieren, dass RFC_M mit der ·Testbarkeit und ·Wartbarkeit von Software zusammenhängt. Folgende Unterhypothesen werden untersucht:

1. RFC_M korreliert positiv mit $\text{NUMISS}(\text{B})_\text{M}$.
2. RFC_M korreliert positiv mit BUGRATE_M.
3. RFC_M korreliert negativ mit BMI_M.
4. RFC_M korreliert negativ mit $\text{ENDRATE}(\text{B})_\text{M}$.
5. RFC_M korreliert negativ mit $\text{IMPROVERATE}_\text{M}$.

Hypothese AP4 Ryder [80:148] beobachtet in Bezug auf HASKELL moderate bis starke Korrelationen (zwischen $r = 0,47$ und $r = 0,57$) des Ausgangsgrades von Funktionen mit der Anzahl von Fehlerkorrekuren. Daher wird die Hypothese:

- FANOUT_F korreliert stark positiv mit $\text{NUMISS}(\text{B})_\text{F}$.

gegen die Nullhypothese $r \leq 0,5$ getestet.

Hypothese AP5 Für ·Instabilität unterstellt Spinellis [86:348] einen positiven Zusammenhang mit der Häufigkeit von Programmänderungen, was mit den beiden Varianten von INST_M wie folgt operationalisiert wird:

1. INSTF_M korreliert positiv mit $\text{STARTRATE}_\text{M}$.
2. INSTM_M korreliert positiv mit $\text{STARTRATE}_\text{M}$.

Hypothese AP6 Hopkins und Hatton [44:8] berichten eine wider Erwarten negative Korrelation der maximalen Verschachtelung von `if`-Anweisungen mit der Fehleranzahl. Daher lauten die beiden Unterhypothesen:

1. MaxCase$_{\mathrm{F}}$ korreliert negativ mit NumIss(B)$_{\mathrm{F}}$.
2. MaxNest$_{\mathrm{F}}$ korreliert negativ mit NumIss(B)$_{\mathrm{F}}$.

Für diese beiden Forschungshypothesen ist die Nullhypothese, dass kein negativer Zusammenhang besteht, das heißt $r \geq 0$.

4.3.1.3 Hypothesen über Trennschärfe

Hypothese T1 In *Program Development Using Erlang - Programming Rules and Conventions* wird empfohlen, `case`, `if` und `receive` um nicht mehr als zwei Ebenen zu verschachteln [26:23], daher werden folgende Unterhypothesen formuliert:

1. Ab einem Wert ≥ 3 von MaxCase$_{\mathrm{FM}}$ gibt es deutlich mehr Fehler.[87]
2. Ab einem Wert ≥ 3 von MaxNest$_{\mathrm{FM}}$ gibt es deutlich mehr Fehler.

Hypothese T2 Auf Grund der Empfehlung in *Program Development Using Erlang - Programming Rules and Conventions*, die Zeilenanzahl von Modulen auf 400 zu beschränken [26:23], wird postuliert:

- Ab 400 Zeilen enthält ein Modul deutlich mehr Fehler.

Hypothese T3 Moores [69:48] ermittelt für Prolog-Programme einen Schwellwert bei 30 ± 5 Regeln unterschiedlichen Namens oder Stelligkeite, oberhalb dessen deutlich mehr Fehler auftreten.[88] Da Erlang und Prolog einander syntaktisch ähneln und beide deskriptive Programmiersprachen sind, werden folgende Hypothesen übertragen:

- Ab etwa 30 Funktionen enthält ein Modul deutlich mehr Fehler.
- Ab etwa 40 Funktionen enthält ein Modul deutlich mehr Fehler.

[87] Zur Bedeutung von „deutlich mehr Fehler“, siehe Abschnitt 4.3.5, S. 86.
[88] t-Test, $p = 0,006$

Hypothese T4 Aus einer weiteren Empfehlung in *Program Development Using Erlang - Programming Rules and Conventions* [26:23] wird folgende Hypothese abgeleitet:

- Ab etwa 15 bis 20 Zeilen enthält eine Funktion deutlich mehr Fehler.

Hypothese T5 Marinescu und Lanza [60:28] mutmaßen für verschiedene Maße, darunter AvgLOC$_{\mathrm{M}}$, dass $1,5 \cdot (\bar{x} + s)$ ein sinnvoller Schwellwert für die Fehlerträchtigkeit von Programmkomponenten ist. Dem liegt die Annahme der Normalverteilung der Maße zugrunde. Wie Tabelle D.2, S. 135 zeigt, ist das Maß deutlich nicht normalverteilt. Daher wird der Schwellwert zunächst aus dem Median und der mittleren Abweichung vom Median gebildet, zum Vergleich wird jedoch auch der auf unzutreffenden Annahmen beruhende Schwellwert untersucht. Die beiden Varianten der Hypothese lauten:

1. Für AvgLOC$_{\mathrm{M}}$ treten ab dem Schwellwert $1,5 \cdot (\tilde{x}_{0,5} + \tilde{d}_{0,5})$ deutlich mehr Fehler auf.
2. Für AvgLOC$_{\mathrm{M}}$ treten ab dem Schwellwert $1,5 \cdot (\bar{x} + s)$ deutlich mehr Fehler auf.

Hypothese T6 Analog zu Hypothese T5 wird postuliert [nach 60:29]:

1. Für AvgCalls$_{\mathrm{M}}$ treten ab dem Schwellwert $1,5 \cdot (\tilde{x}_{0,5} + \tilde{d}_{0,5})$ deutlich mehr Fehler auf.
2. Für AvgCalls$_{\mathrm{M}}$ treten ab dem Schwellwert $1,5 \cdot (\bar{x} + s)$ deutlich mehr Fehler auf.

4.3.2 Hinweise zu den Grafiken

Da es teilweise verschiedene Definitionen der verwendeten Grafikarten gibt, seien kurz die im Folgenden verwendeten Konventionen dargelegt.

4.3.2.1 Boxplot

Die Box reicht vom unteren bis oberen Quartil ($\tilde{x}_{0,75}$ und $\tilde{x}_{0,25}$). Als Ausreißer gelten Werte ab 1,5-fachem Quartilsabstand $d_Q = \tilde{x}_{0,75} - \tilde{x}_{0,25}$ [89:73] von Ober- oder Untergrenze der Box. Die „Antennen“ reichen bis zum kleinsten bzw. größten Nicht-Ausreißer und sind dort ***nicht*** durch horizontale Striche abgeschlossen. Der Median ist als horizontale Linie, das arithmetische Mittel als Stern dargestellt.

4.3.2.2 Streudiagramm

Die Achsen beginnen bei Null, auch wenn diese nicht gekennzeichnet ist. Die Punkte sind halbtransparent dargestellt, so dass einzelne Punkte grau und mehrere übereinander liegende Punkte dunkler erscheinen. Die Randverteilungen sind durch achsenparallele Boxplots dargestellt. Gestrichelte achsenparallele Geraden im Streudiagramm zeigen die Ausreißergrenzen der Boxplots an. Gepunktete Geraden zeigen die Grenzen für ***Extremwerte*** (dreifacher Quartilsabstand vom unteren beziehungsweise oberen Quartil [89:84]). Am unteren Rand ist jeweils die Stichprobengröße n angegeben.

4.3.3 Kenngrößen der einzelnen Maße

In diesem Abschnitt werden die an EJABBERD erhobenen Messwerte in Bezug auf Lage und Streuung charakterisiert. Zunächst werden die ermittelten Kenngrößen der internen Maße vorgestellt, anschließend die der externen Maße.

4.3.3.1 Kenngrößen der internen Maße

Tabelle D.1, S. 134 und Tabelle D.2, S. 135 zeigen die ermittelten zentralen Momente der internen Maße für Funktionen und Module. Man sieht, dass die Verteilungen aller Maße auf Funktionsebene und fast aller Maße auf Modulebene deutlich rechtsschief und stark gewölbt sind [89:82f.]. Dies stimmt mit den Ergebnissen von Ryder [80:342ff.] für HASKELL überein und entspricht auch Beobachtungen an prozeduralen und objektorientierten Programmen [66:204] [90:92]. Von den Maßen, die bei der Hypothesenprüfung noch betrachtet werden, weisen nur die Maße MAXCASE$_M$, MAXNEST$_M$, INSTF$_M$ und INSTM$_M$ zentrale Momente auf, die in etwa

denen einer Normalverteilung entsprechen [64:66]. Auf Grund der starken Asymmetrie der meisten Maße, mit vielen „Ausreißern“, ist insgesamt als Lagemaß der Median $\tilde{x}_{0,5}$ besser geeignet als das arithmetische Mittel [17:82], und entsprechend die mittlere absolute Abweichung vom Median $\tilde{d}_{0,5}$ als Streuungsmaß [89:74].

Tabelle D.3, S. 136 zeigt Minimum, Maximum, Quartile und Ausreißer-Grenzen der internen Maße für Funktionen. Ausreißer (im Boxplot-Sinn) nach unten kommen nicht vor (die Grenzen sind kleiner als das theoretische und empirische Minimum). Während bei einigen Maßen sehr hohe Extremwerte vorkommen, liegt der Großteil der Werte in einem kleinen Bereich um den Median. Die mittlere Abweichung vom Median ist überwiegend deutlich kleiner als ein Zehntel des Maximums. Tabelle D.4, S. 137 zeigt die gleichen Kennwerte für Module. Auch hier kommen Ausreißer nach unten nicht vor. Die mittlere Abweichung vom Median ist meist größer als auf Funktionsniveau, etwa ein Zehntel des jeweiligen Maximums.

4.3.3.2 Kenngrößen der externen Maße

Tabelle D.5, S. 138 zeigt die für EJABBERD festgestellten zentralen Momente der externen Maße auf Funktionsniveau. Auch diese Maße sind überwiegend deutlich rechtsschief und stark gewölbt. Die Ausnahme stellen die Maße für die mittlere Bearbeitungszeit von Fällen, $\text{MEDITT}(*)_{\text{M}}$, dar. Das gleiche Bild ergibt sich für die externen Maße auf Modulebene, Tabelle D.6, S. 139.

Tabelle D.7, S. 140 und Tabelle D.8, S. 141 zeigen Minimum, Maximum, Quartile und Ausreißer-Grenzen der externen Maße für Funktionen und Module. Es fällt auf, dass bis auf die mittlere Fallbearbeitungszeit alle Maße im Bereich zwischen dem unteren und oberen Quartil durchgängig den Wert Null annehmen. Dies resultiert aus der im Vergleich zur Anzahl der Module und Funktionen geringen Anzahl an ·Fällen, die diesen zugeordnet werden konnten (siehe Abschnitt 4.2.3, S. 69).

4.3.4 Zusammenhänge zwischen den Maßen

Im Folgenden werden die in dieser Studie beobachteten Zusammenhänge der in den Hypothesen vorkommenden Maße erläutert, zunächst für die internen Maße, gefolgt von den externen.

Tabelle 4.4 – Korrelationsmatrix der internen Maße für Funktionen bei EJABBERD. Erhebliche negative Koeffizienten sind schwarz hinterlegt, starke positive Korrelationen sind **fett** gesetzt und sehr starke zusätzlich grau hinterlegt. Nicht hervorgehobene Korrelationskoeffizienten sind unerheblich.

	WMC(Cyc)_M	CBO_M	RFC_M	InstF_M	InstM_M	MaxCase_M	MaxNest_M	NumFun_M	AvgLOC_M	AvgCalls_M
CBO_M	0.14	-								
RFC_M	**0.92**	0.15	-							
InstF_M	**0.20**	-0.29	**0.25**	-						
InstM_M	0.09	-0.07	0.14	**0.72**	-					
MaxCaseM_M	**0.58**	0.19	**0.62**	0.18	0.09	-				
MaxNestM_M	**0.57**	0.17	**0.61**	**0.21**	0.10	**0.92**	-			
NumFun_M	**0.82**	0.15	**0.92**	0.17	0.06	**0.50**	**0.49**	-		
AvgLOC_M	**0.32**	0.02	**0.30**	0.16	0.16	**0.38**	**0.37**	0.11	-	
AvgCalls_M	0.03	**0.47**	0.06	-0.48	-0.32	**0.24**	**0.23**	-0.02	**0.21**	-
LOC_M	**0.96**	0.12	**0.96**	**0.22**	0.10	**0.61**	**0.60**	**0.89**	**0.38**	0.04

4.3.4.1 Zusammenhänge zwischen den internen Maßen

Ein Aspekt des Kriteriums ·Verbesserungsvalidität ist die Frage, ob ein Maß nützlicher ist als das simpelste Maß, LOC_FM. Wiederholt wird in der Literatur festgestellt, dass viele Softwaremaße stark positiv mit LOC_FM korrelieren [36:654] [44:6]. So überrascht es nicht, dass dies auch in dieser Studie der Fall ist, wie Tabelle 4.4 zeigt.

Die Maße WMC(Cyc)_M, RFC_M und NumFun_M korrelieren sehr stark positiv ($r > 0,80$) mit der Zeilenanzahl und miteinander. Es ist zu erwarten, dass beide Maße zu sehr ähnlichen Ergebnissen bei der Hypothesenprüfung führen. Falls nicht alle Ausdrücke eines Programms in eine Zeile geschrieben werden, können neue ·Prädikatknoten im Kontrollflussgraphen (deren Anzahl WMC(Cyc)_M im wesentlichen angibt) nur durch das Hinzufügen von Zeilen entstehen; gleiches gilt für neue Funktionen oder Funktionsaufrufe (deren Anzahl NumFun_M beziehungsweise RFC_M angeben). Der Korrelation liegt also ein kausaler Zusammenhang zugrunde.

MaxCase_M und MaxNest_M korrelieren ebenfalls stark positiv ($r \approx 0,60$) mit der Zeilenanzahl. Dies ist darauf zurückzuführen, dass bei „normaler“ Formatie-

Tabelle 4.5 – Korrelationsmatrix der internen Maße für Funktionen bei EJABBERD. Erhebliche negative Koeffizienten sind schwarz hinterlegt, starke positive Korrelationen sind **fett** gesetzt und sehr starke zusätzlich grau hinterlegt. Nicht hervorgehobene Korrelationskoeffizienten sind unerheblich.

	FANOUT$_F$	MAXCASE$_F$	MAXNEST$_F$
MAXCASE$_F$	**0.58**	-	
MAXNEST$_F$	**0.66**	**0.89**	
LOC$_F$	**0.76**	**0.58**	**0.59**

rung jede Verschachtelungsebene weitere Zeilen belegt.

Nicht wesentlich mit LOC$_M$ korreliert ($r \leq 20$) sind die Anzahl der mit einem Modul gekoppelten Module, CBO$_M$, das ·Instabilitäts-Maß INSTM$_M$ und die durchschnittliche Anzahl von Funktionsaufrufen pro Funktion eines Moduls, AVGCALLS$_M$. Dies macht sie interessant als Maße, die wesentlich etwas anderes als die Größe eines Moduls widerspiegeln. AVGCALLS$_M$ und CBO$_M$ sind moderat positiv korreliert ($r = 0,47$), da sie beide aus der Anzahl der Funktionsaufrufbeziehungen abgeleitet sind. Aus dem gleichen Grund sind sie mit den beiden ·Instabilitäts-Maßen, bei deren ·Messfunktionen die Anzahl der Aufrufbeziehungen im Nenner steht, moderat negativ korreliert ($-0,48 \leq r \leq -0,29$).

Auf der Funktionenebene korrelieren alle betrachteten Maße stark ($r > 0,5$) mit der der Zeilenanzahl und miteinander, wie Tabelle 4.5 zeigt. FANOUT$_F$ korreliert am stärksten mit LOC$_F$, weil mit zunehmender Zeilenanzahl die Anzahl der Funktionsaufrufe steigen muss, wenn nicht hauptsächlich arithmetische oder logische Operationen ohne Funktionsaufrufe ausgeführt werden. Für MAXCASE$_F$ und MAXNEST$_F$ gilt das oben für die Modulebene Gesagte.

Validierungskriterium ·Faktorenunabhängigkeit ·Faktorenunabhängigkeit bezeichnet die Forderung, dass die Komponenten abgeleiteter Maße voneinander unabhängig sein sollen (siehe Abschnitt 2.3.1.1, S. 35). Dies wird nun anhand der linearen Korrelation für die in den Hypothesen verwendeten Maße untersucht.

CBO$_M$ kann als abgeleitetes Maß aus der Anzahl der fortführenden und der hinführenden Modulkopplungen, OUTMODS$_M$ und INMODS$_M$, aufgefasst werden. Die beiden Maße weisen eine signifikante Korrelation von $r = 0.31$[89] auf. Dies ist nicht

[89] Konfidenzintervall: $[0.27, 0.34]$

wesentlich auf einen gemeinsamen Zusammenhang mit der Zeilenanzahl zurückzuführen, die mit InMods_M geringfügig negativ korreliert ist ($r = -0,04$ im Konfidenzintervall $[-0,08, 0,00]$), mit OutMods_M dagegen moderat positiv ($r = 0,40$ im Konfidenzintervall $[0,38, 0,44]$).

Das Maß RFC_M ist als Summe aus NumFun_M und CallsOut_M definiert. Diese beiden Maße sind mit $r = 0,86$[90] stark positiv korreliert, was nicht überrascht, da mit der Anzahl der Funktionen im Modul auch die Anzahl der Funktionsaufrufe aus diesem Modul steigen muss, sofern diese Funktionen nicht nur triviale Berechnungen ohne externe Funktionen durchführen.

Das ·Instabilitäts-Maß InstM_M ist von den Maßen OutMods_M und InMods_M abgeleitet, die bereits oben betrachtet wurden. Die Variante InstF_M dagegen ist von OutFuns_M und InFuns_M abgeleitet. Diese weisen eine ähnliche Korrelation mit LOC_M auf wie Modul-Pendants: OutFuns_M ist stark positiv korreliert ($r = 0.91$, Konfidenzintervall: $[0.90, 0.91]$), InFuns_M fast unkorreliert ($r = 0.023$, Konfidenzintervall: $[-0.01, 0.06]$).

4.3.4.2 Zusammenhänge zwischen den externen Maßen

Tabelle 4.6 auf der nächsten Seite zeigt die linearen Zusammenhänge der externen Maße auf Modulebene für EJABBERD. Es fällt auf, dass EndRate_M bei den von FEHLERN[91] betroffenen Modulen stark ($r = 0,63$) mit $\text{NumIss(B)}_\text{M}$ korreliert. Bijlsma [8:39] definiert $\text{EndRate(B)}_\text{M}$ als Maß der „Projektproduktivität“ und interpretiert hohe Werte als Zeichen effizienter Fallbearbeitung. Jedoch muss auf Grund der Beobachtungen an EJABBERD nun festgestellt werden, dass das Maß $\text{EndRate(B)}_\text{M}$ anscheinend wesentlich vom FEHLER-Gehalt der Module beeinflusst wird. Dies ist plausibel, da nur dann viele FEHLER gelöst werden können, wenn überhaupt viele FEHLER vorhanden sind. $\text{EndRate(B)}_\text{M}$ erfasst somit weniger die ·Wartbarkeit der Software als ihre FEHLER-Trächtigkeit. Dies wird Auswirkungen auf alle Hypothesen haben, in denen $\text{EndRate(B)}_\text{M}$ als Maß für ·Wartbarkeit verwendet wird.

Desweiteren bestehen bemerkenswerte schwache bis moderate ***negative*** Zusammenhänge ($-0,43 \leq r \leq 0,20$) zwischen der mittleren Bearbeitungszeit von Fäl-

[90] Konfidenzintervall $[0,85, 0,97]$
[91] Siehe Abschnitt 2.2.4, S. 31.

Tabelle 4.6 – Korrelationsmatrix der internen Maße für Funktionen. Erhebliche negative Koeffizienten sind schwarz hinterlegt, starke positive Korrelationen sind **fett** gesetzt und sehr starke zusätzlich grau hinterlegt. Nicht hervorgehobene Korrelationskoeffizienten sind unerheblich.

	NumIss(B)$_M$	BugRate$_M$	MedITT(B)$_M$	BMI$_M$	StartRate(B)$_M$	EndRate(B)$_M$
BugRate$_M$	**0.55**	-				
MedITT(B)$_M$	-0.20	0.00	-			
BMI$_M$	**0.34**	**0.26**	-0.28	-		
StartRate(B)$_M$	**0.76**	**0.43**	-0.43	0.23	-	
EndRate(B)$_M$	**0.74**	**0.40**	-0.37	**0.39**	**0.73**	-
ImproveRate$_M$	0.15	0.01	-0.03	**0.56**	0.10	0.07

len, MedITT(B)$_M$, und der Gesamtanzahl der Fälle beziehungsweise Maßen, die stark mit dieser korrelieren (in Tabelle 4.6 schwarz hinterlegt). Die Fallbearbeitungszeit war also im Mittel am ***kürzesten*** bei den Modulen mit den ***meisten*** Fehlern. Mögliche Erklärungen sind einerseits, dass die hohen Fehler-Zahlen einen großen Anteil leicht behebbarer Fehler widerspiegeln, und andererseits, dass viele Fehler vor allem bei häufig verwendeten und besonders wichtigen Modulen vorkommen und dort mit höherer Priorität bearbeitet werden (siehe Abschnitt 2.3.4.2, S. 40 für verzerrende Einflüsse auf die Verteilung der Fehler über die Programmkomponenten). Diese Vermutungen müssten in weiteren Untersuchungen überprüft werden.

4.3.5 Verwendete Verfahren

Die statistischen Verfahren wurden den Empfehlungen in IEEE [46:18f.] entsprechend ausgewählt. Alle Berechnungen werden mit dem Statistiksystem R durchgeführt (siehe Anhang B.1, S. 126). Die Ergebnisse sind auf zwei Nachkommastellen gerundet.

Das ·Validierungskriterium ·Assoziation wird mit dem Korrelationskoeffizienten von Bravais-Pearson [89:131] (im Folgenden nur noch kurz mit r bezeichnet)

überprüft,[92] wobei die Entscheidung über die Hypothesen anhand des 95-Prozent-Konfidenzintervalls erfolgt.

·Parallele Veränderung wird mit dem Rangkorrelationskoeffizienten von Spearman [89:126] (im Folgenden nur noch kurz R) untersucht. Wenn das Kriterium ·Assoziation bereits erfüllt ist, gilt das Kriterium ·Parallele Veränderung ebenfalls als erfüllt; die Umkehrung gilt nicht.

Die Überprüfung des Kriteriums ·Trennschärfe erfolgt durch Pearsons χ^2-Statistik. Zugunsten der Vergleichbarkeit werden die χ^2-Ergebnisse als Beträge des Φ-Koeffizienten normiert im Intervall $[0, 1]$ dargestellt [89:112]. Für unabhängige Merkmale ist $\Phi = 0$; je größer $\bar{\Phi}$ ist, desto stärker ist der Zusammenhang. Überprüft werden nur die in der Literatur genannten Schwellwerte. Auf der Grundlage der Daten mehrerer Projekte könnten bessere Schwellwerte gesucht werden, was auf der Grundlage eines einzelnen Projekts nicht sinnvoll erscheint. Die Wahrscheinlichkeit β des ·Fehlers 2. Art wird nach Cohen [18] berechnet.

Die Anzahl der Fehler wird durch $\text{NumIss(B)}_{\text{FM}}$ gemessen. Als Grenzen für „deutlich mehr“ Fehler wurden die Ausreißer- und Extremwertgrenzen nach Boxplot-Konvention gewählt. Wie Tabelle D.7, S. 140 und Tabelle D.8, S. 141 zeigen, liegen diese Grenzen alle bei Null, bei einer mittleren Abweichung vom Median bei $0,1$ beziehungsweise $0,4$ (auf zwei Nachkommastellen gerundet). Daher wurden für die Kontingenztafeln Null oder mehr Fehler als „deutlich mehr“ gewertet.

Mit diesen einfachen statistischen Verfahren wird lediglich der ***lineare*** Zusammenhang von Maßen erfasst, beziehungsweise die ·Trennschärfe nur in Bezug auf ***vorgegebene*** Schwellwerte überprüft.

4.3.6 Überprüfung der Hypothesen

4.3.6.1 Hypothese AP1

Unterhypothese AP1.1 Abb. 4.7 auf der nächsten Seite (linkes Diagramm) setzt für alle Versionen aller Module ihren jeweiligen $\text{WMC(Cyc)}_{\text{M}}$-Wert mit der

[92] Dies erfolgt gemäß Motulsky [70:300] trotz der meist deutlich nicht-normalen Verteilungen in der Annahme, dass die Stichproben groß genug sind, um gestützt auf den Zentralen Grenzwertsatz dennoch sinnvolle Ergebnisse zu erhalten.

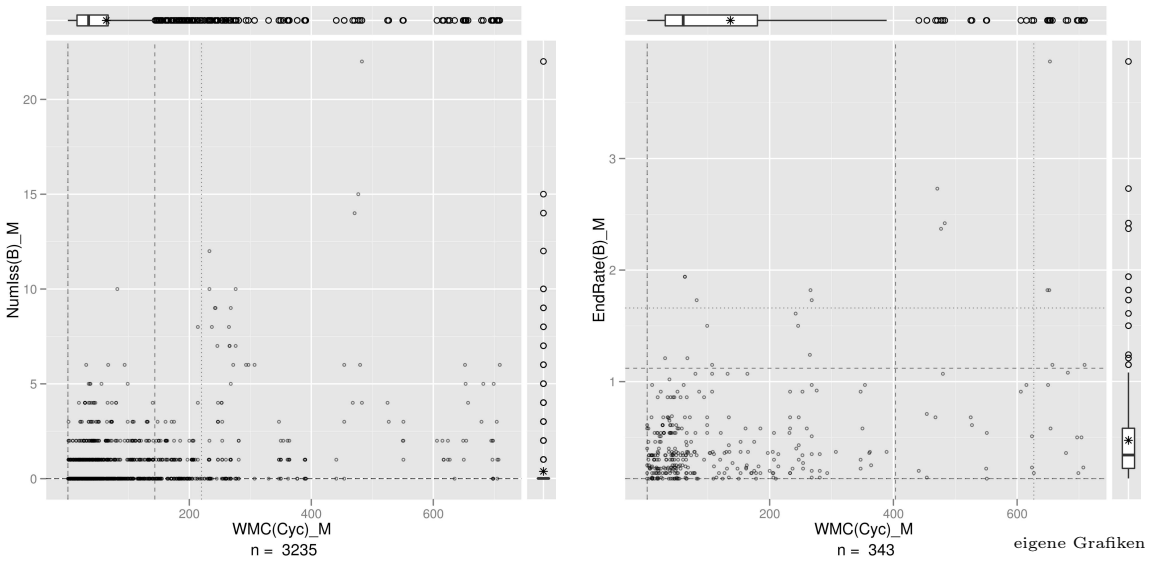


Abbildung 4.7 – Streudiagramme für $\text{WMC}(\text{CYC})_\text{M}$ und $\text{NUMISS}(\text{B})_\text{M}$ (links) und $\text{ENDRATE}(\text{B})_\text{M}$ (rechts).

Anzahl der FEHLER-Fälle $\text{NUMISS}(\text{B})_\text{M}$ in Beziehung, die die jeweilige Version des Moduls betreffen.

An den beiden Boxplots wird die erhebliche Rechtsschiefe der Randverteilungen ersichtlich, die auch bei den anderen Maßen zu beobachten sein wird: Module ohne oder mit nur wenigen FEHLERN (wohlgemerkt über ihre gesamte Lebenszeit!) und mit einem $\text{WMC}(\text{CYC})_\text{M}$-Wert unter 200 überwiegen deutlich. Die Box des Boxplots für $\text{NUMISS}(\text{B})_\text{M}$ ist kaum zu erkennen, da mehr als die Hälfte der Module überhaupt keine FEHLER hat. Auch mehrere deutlich „schwerere“ Module haben weniger als fünf FEHLER. Größere $\text{NUMISS}(\text{B})_\text{M}$-Werte treten dagegen hauptsächlich bei Modulen über 200 $\text{WMC}(\text{CYC})_\text{M}$ auf.

Insgesamt beträgt der lineare Zusammenhang $r = 0.38$,[93] womit die Nullhypothese zugunsten von Hypothese AP1.1 verworfen wird.

Unterhypothese AP1.2 Abb. 4.8, S. 91 (linkes Diagramm) zeigt die Werte von $\text{WMC}(\text{CYC})_\text{M}$ aller Versionen aller Module mit dem dazugehörigen Wert von BUGRATE_M, dem Anteil von FEHLERN an allen Fällen, die die jeweilige Modulversion betreffen.

[93] Konfidenzintervall: $[0.35, 0.41]$

Da mehr als die Hälfte der Module keine FEHLER hat, ist auch die Box für die Verteilung von BUGRATE_M auf den Nullpunkt zusammengestaucht. Es dominieren somit die Modulversionen, die keine oder nur sehr wenige FEHLER oder andere Fälle enthalten. Angesichts dessen, dass die meisten Module nicht mehr als vier FEHLER haben (siehe Abb. 4.7, S. 88), deuten die Häufungen bei $0,5$ und 1 für BUGRATE_M auf viele Module mit einem oder zwei FEHLERN bei ebensovielen anderen Fällen hin.

Die höchsten FEHLER-Anteile treten bei den Modulen mit WMC(CYC)_M-Werten unterhalb von 200 auf, während im mittleren Bereich des Spektrums kaum und unter den „schwersten“ Modulen überhaupt keine Anteile über 0,5 vorkommen. Dies stimmt mit den Beobachtungen anderer Autoren überein, dass größere Module relativ weniger Fehler enthalten, was auf größere Aufmerksamkeit und Sorgfalt bei der Bearbeitung großer Module zurückgeführt wird [vgl. 44:9f.].

Insgesamt ergibt sich so über den gesamten Wertebereich nur ein linearer Zusammenhang von $r = 0,17$, was nicht ausreicht, um die Nullhypothese für das Kriterium ·Assoziation zu verwerfen. Allerdings ist der Rangkorrelationskoeffizient signifikant mit $p = 0.00$ und mit $R = 0,29$ schwach, aber für das Kriterium ·Parallele Veränderung ausreichend.

Unterhypothese AP1.3 Mit der mittleren Bearbeitungszeit von FEHLERN in Stunden, $\text{MEDITT(B)}_\text{M}$, weist WMC(CYC)_M eine signifikante Rangkorrelation auf ($p = 0.00$). Das Ausmaß der Korrelation, $R = 0,14$, ist jedoch als unerheblich einzustufen. Da der Pearsonsche Korrelationskoeffizient nicht signifikant ist ($p = 0,93$), wird die Nullhypothese sowohl in Bezug auf das Kriterium ·Assoziation als auch ·Parallele Veränderung beibehalten.

Unterhypothese AP1.4 Ebenfalls vernachlässigbar ist der Zusammenhang von WMC(CYC)_M mit dem ·Backlog Management Index BMI_M. Die Verteilung der Wertepaare weist kein erkennbares Muster auf (siehe Abb. D.1, S. 136 im Anhang). Allerdings zeigte sich hier entgegen dem postulierten negativen Zusammenhang mit $r = 0,17$[94] ein positiver Zusammenhang. Die Nullhypothese wird zuungunsten von Hypothese AP1.4 beibehalten.

[94] Konfidenzintervall: $[0.14, 0.21]$

Unterhypothese AP1.5 Mit EndRate(B)$_M$ zeigt das Maß WMC(Cyc)$_M$ einen positiven Zusammenhang, wo ein negativer vermutet wurde. Abbildung 4.7, S. 88 (rechtes Diagramm) zeigt die WMC(Cyc)$_M$-Werte aller Modulversionen, die Fehler enthalten, mit der entsprechenden durchschnittlichen Anzahl gelöster Fehler pro Monat während der Lebenszeit der Version. Die Verteilung ähnelt der im linken Diagramm. EndRate(B)$_M$ ist ebenfalls stark mit NumIss(B)$_M$ korreliert.

Der lineare Zusammenhang mit WMC(Cyc)$_M$ ist mit $r = 0,34$[95] etwas schwächer als der von NumIss(B)$_M$. Das Ergebnis deutet darauf hin, dass die Grundannahme, die zur Formulierung von Hypothese AP1.5 führte – dass EndRate(B)$_M$ positiv mit Wartbarkeit zusammenhängt –, zu einfach ist. Der Einfluss der Fehler-Anzahl scheint zu überwiegen. Hypothese AP1.5 wird zurückgezogen, da sie sachlich nicht gerechtfertigt erscheint. Das Ergebnis ist aber auch nicht mit der Nullhypothese vereinbar. Weitere Untersuchungen zur Beziehung von EndRate(B)$_M$ und ·Wartbarkeit erscheinen angebracht.

Unterhypothese AP1.6 Das Maß für den Anteil von Verbesserungen an allen Fällen, ImproveRate$_M$ (Abb. 4.8 auf der nächsten Seite, rechtes Diagramm), zeigt für Module im unteren Wertebereich von WMC(Cyc)$_M$ die gleichen Häufungen bei 0, 0,5 und 1 wie BugRate$_M$, wobei die Randverteilung wesentlich gleichmäßiger ist, wie die über zwei Drittel des Wertebereichs gestreckte Box zeigt. Im ganzen Wertebereich von WMC(Cyc)$_M$ kommen sowohl minimale wie mittlere und maximale ImproveRate$_M$-Werte gleichermaßen vor.

Entgegen des hypothetischen negativen Zusammenhangs ergibt sich ein positiver, wenn auch vernachlässigbarer Korrelationskoeffizient von $r = 0,10$.[96] Die Rangkorrelation ist signifikant mit $p = 0,00$, aber wider Erwarten erheblich positiv: $R = 0,22$. Dies führt zuungunsten von Hypothese AP1.6 zur Beibehaltung der Nullhypothese, dass kein erheblicher negativer Zusammenhang besteht.

Dass in Bezug auf WMC(Cyc)$_M$ umfangreichere Module nicht wie vermutet geringere Verbesserungsraten aufweisen, lässt sich zum Teil mit der geringeren Fehlerdichte erklären – Module mit weniger zu behebenden Fehlern bieten relativ mehr Raum für Verbesserungen. Insgesamt muss die Hypothese AP1.6 verworfen werden.

[95] Konfidenzintervall: $[0.24, 0.43]$
[96] Konfidenzintervall: $[0.04, 0.16]$

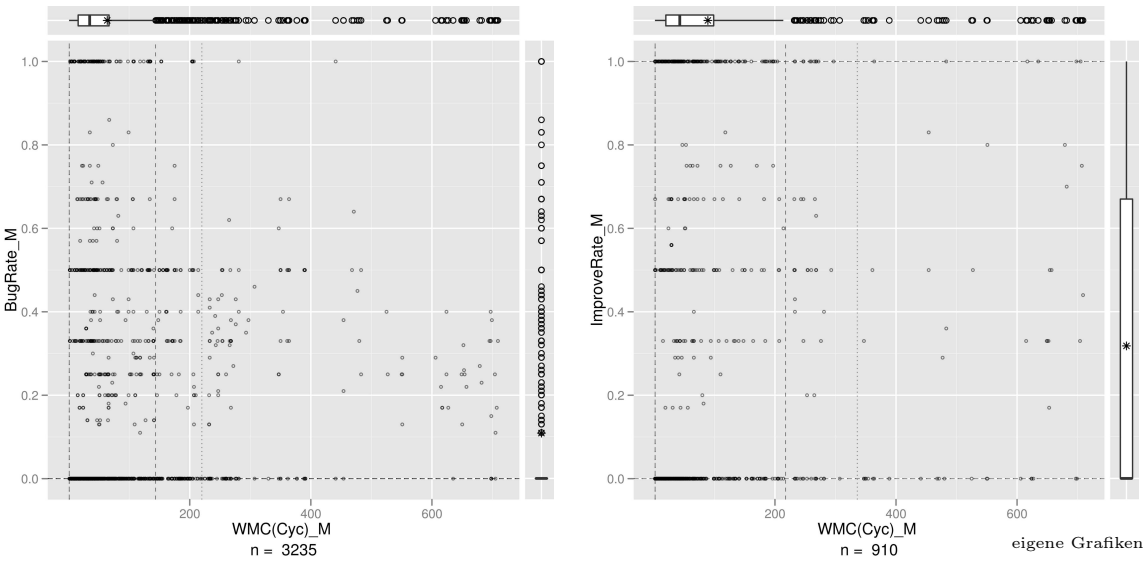


Abbildung 4.8 – Streudiagramme für $\text{WMC}(\text{CYC})_\text{M}$ und BUGRATE_M (links) und $\text{IMPROVERATE}_\text{M}$ (rechts).

Zusammenfassung Insgesamt haben sich die Unterhypothesen von AP1 als miteinander unvereinbar herausgestellt, was hauptsächlich daran liegt, dass sich zwischen den externen Maßen Zusammenhänge zeigen, mit denen bei Aufstellung der Hypothesen nicht gerechnet wurde. Eine summarische Aussage zu Hypothese AP1 kann daher nicht getroffen werden.

Festgestellt wurde, dass $\text{WMC}(\text{CYC})_\text{M}$ moderat positiv ($r = 0,38$) mit der Anzahl der FEHLER und wider Erwarten mit der Lösungsrate von FEHLERN ($r = 0,34$) korreliert. Zudem besteht eine schwache Rangkorrelation mit dem FEHLER-Anteil an den Fällen ($R = 0,29$) sowie mit dem VERBESSERUNGS-Anteil an gelösten Fällen ($R = 0,22$).

4.3.6.2 Hypothese AP2

Unterhypothesen AP2.1 und AP2.2 In Bezug auf die Maße BUGRATE_M und $\text{NUMISS}(\text{B})_\text{M}$ weist CBO_M praktisch keinen linearen Zusammenhang auf ($r = 0,05$ beziehungsweise $r = 0,06$), weshalb die Nullhypothese in beiden Fällen beibehalten wird.

Während die Verteilung mit BUGRATE_M keinerlei ausgeprägten Zusammenhang erkennen lässt (siehe Abb. D.2, S. 142), weist die Verteilung mit $\text{NUMISS}(\text{B})_\text{M}$

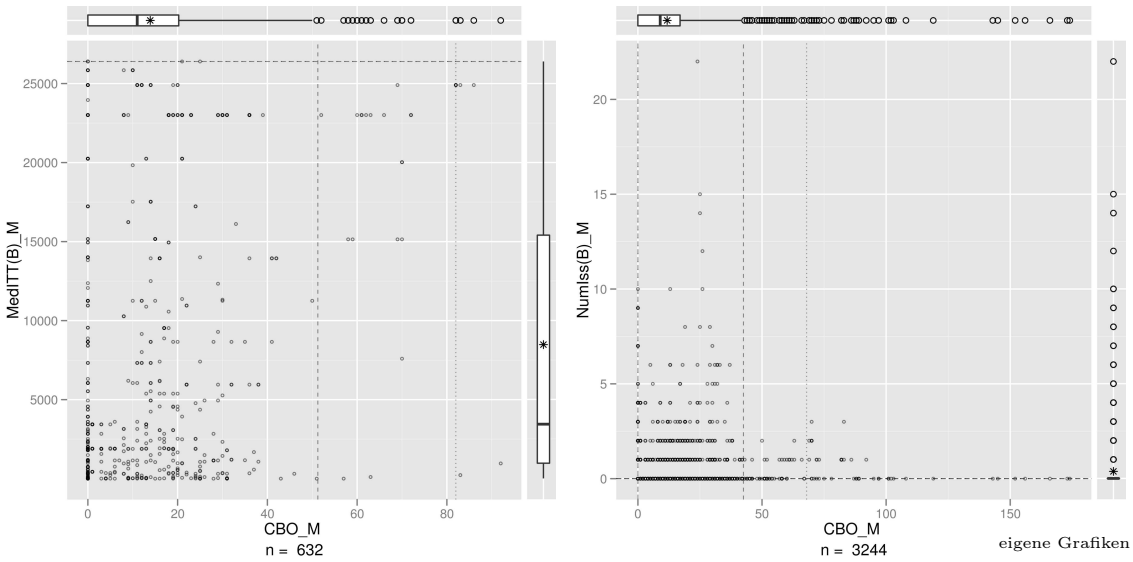


Abbildung 4.9 – Streudiagramme für $\mathrm{CBO_M}$ und $\mathrm{MEDITT(B)_M}$ (links) beziehungsweise $\mathrm{NUMISS(B)_M}$ (rechts)

(Abb. 4.9, rechtes Diagramm) bei einem $\mathrm{CBO_M}$-Wert um 25 eine ausgeprägte Häufung besonders FEHLER-trächtiger Module auf, während bei den besonders stark gekoppelten Modulen (etwa ab $\mathrm{CBO_M}$-Wert 50) eine leichte Häufung relativ FEHLER-trächtigerer Module bei einem Wert um 70 zu erkennen ist. Weitere Untersuchungen könnten die Besonderheiten dieser Module näher betrachten.

Unterhypothese AP2.3 Abbildung 4.9, S. 92 zeigt den Zusammenhang des Kopplungsmaßes $\mathrm{CBO_M}$ mit der mittleren Bearbeitungsdauer von FEHLERN, $\mathrm{MEDITT(B)_M}$, wobei jeder Punkt einem von FEHLERN betroffenen Modul in einer bestimmten Version entspricht.

Während im unteren Bereich von $\mathrm{CBO_M}$ (unter 40) alle Bearbeitungszeiten von beinahe Null bis über 25000 Stunden (das heißt fast drei Jahre) vorkommen, zeigt sich im oberen Bereich (über 40) ein deutlicher positiver Trend.

Zusammen ergibt sich ein schwacher Pearsonscher Korrelationskoeffizient von $r = 0,22$, der auf Grund des Konfidenzintervalls $[0.14, 0.29]$ nicht zur Ablehnung der Nullhypothese genügt, zumal die Rangkorrelation zwar signifikant von 0 verschieden, aber ebenfalls zu schwach ist: $R = 0,12$. Die Nullhypothese muss beibehalten werden.

Unterhypothesen AP2.4 bis AP2.6 Auch mit den Maßen BMI_M, EndRate_M und $\text{ImproveRate}_\text{M}$ weist CBO_M keinen erkennbaren Zusammenhang auf. Die Korrelationskoeffizienten sind vernachlässigbar ($-0,05 \leq r \leq 0,16$, $0,00 \leq R \leq 0,07$). Die Nullhypothese wird beibehalten.

Zusammenfassung Für das Maß CBO_M konnte keine der Hypothesen bestätigt werden. Mit keinem der angenommenen Indikatoren für ·Wartbarkeit war ein erheblicher positiver oder negativer Zusammenhang festzustellen.

4.3.6.3 Hypothese AP3

Unterhypothese AP3.1 In Abb. 4.10 auf der nächsten Seite ist der Zusammenhang des Maßes für die ·Antwortmächtigkeit, RFC_M, aller Versionen aller Module mit den jeweiligen Fehler-Anzahlen ($\text{NumIss(B)}_\text{M}$, linkes Diagramm) beziehungsweise der monatlichen Durchschnittsanzahl behobener Fehler während der Lebenszeit der Versionen ($\text{EndRate(B)}_\text{M}$, rechtes Diagramm) dargestellt.

Wie oben bereits festgestellt wurde, korrelieren diese beiden externen Maße stark miteinander. Auch ihre gemeinsamen Verteilungen mit RFC_M ähneln sich. Es ist eine sehr starke Häufung von Modulen mit weniger als fünf Fehlern bei RFC_M-Werten unterhalb von 200 zu erkennen. Einen deutlich größeren Fehler-Gehalt weisen Module zwischen 200 und 400 RFC_M auf, während Module mit einem RFC_M-Wert um 500 eher denen am unteren Wertebereich ähneln, allerdings ohne die extreme Häufung von Modulen ohne Fehler.

Insgesamt ergibt sich ein moderater linearer Zusammenhang (der stärkste in dieser Untersuchung) mit $r = 0,39$,[97] so dass die Nullhypothese zugunsten von Hypothese AP3.1 verworfen wird.

Unterhypothesen AP3.2 und AP3.5 Keine ausreichende Pearson-Korrelation weist RFC_M mit den Maßen für den Fehler-Anteil an allen offenen Fällen, BugRate_M, und für den Verbesserungs-Anteil an allen gelösten Fällen, $\text{ImproveRate}_\text{M}$, auf.[98] Daher wird die Nullhypothese bezüglich des Kriteriums

[97] Konfidenzintervall: $[0.36, 0.42]$

[98] $r = 0,18$ für BugRate_M, $r = 0,15$ für $\text{ImproveRate}_\text{M}$.

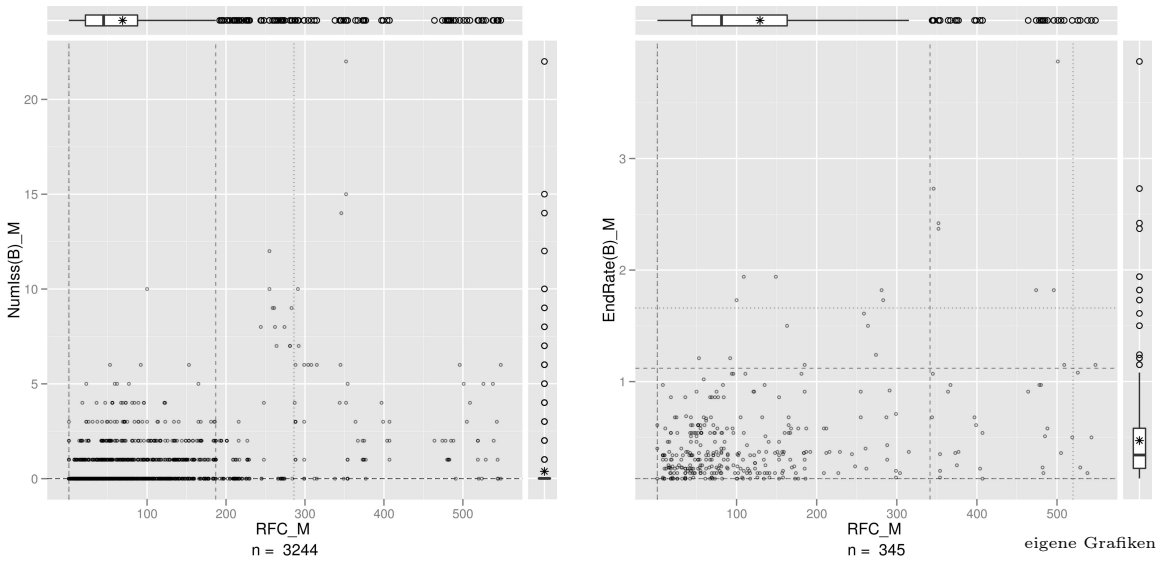


Abbildung 4.10 – Streudiagramme für $\mathrm{RFC_M}$ und $\mathrm{NumIss(B)_M}$ (links) beziehungsweise $\mathrm{EndRate(B)_M}$ (rechts)

·Assoziation beibehalten. Allerdings weisen beide Maße eine ausreichende, schwache Rangkorrelation auf ($R = 0,29$ beziehungsweise $R = 0,25$), so dass die Nullhypothesen unter dem Kriterium ·Parallele Veränderung zugunsten der Hypothesen AP3.2 und AP3.5 verworfen werden. Die gemeinsamen Verteilungen von $\mathrm{BugRate_M}$ und $\mathrm{ImproveRate_M}$ mit $\mathrm{RFC_M}$ gleichen denen mit $\mathrm{WMC(Cyc)_M}$.

Unterhypothese AP3.3 $\mathrm{RFC_M}$ weist einen gegenüber $\mathrm{WMC(Cyc)_M}$ etwas stärkeren positiven Zusammenhang mit dem Maß $\mathrm{BMI_M}$ auf: $r = 0,20$.[99] Die Hypothese einer ***negativen*** Korrelation mit Betrag $> 0,20$ muss verworfen werden. Die Nullhypothese, dass der Betrag des Korrelationskoeffizienten $\leq 0,20$ ist, wird angesichts des Konfidenzintervalls durch das Ergebnis nicht widerlegt und wird daher beibehalten. Die gemeinsame Verteilung von $\mathrm{RFC_M}$ mit $\mathrm{BMI_M}$ gleicht im Übrigen der bei $\mathrm{WMC(Cyc)_M}$ und weist keine bemerkenswerte Regelmäßigkeit auf (siehe Abb. 4.12 auf der nächsten Seite).

Unterhypothese AP3.4 $\mathrm{CBO_M}$ weist auch mit $\mathrm{EndRate(B)_M}$ eine moderate, wider Erwarten positive Korrelation auf: $r = 0,36$.[100] Dass $\mathrm{EndRate(B)_M}$ vermutlich nur zu einem geringen Teil ein Maß für ·Wartbarkeit und die Hypothese

[99] Konfidenzintervall: $[0.16, 0.23]$
[100] Konfidenzintervall: $[0.27, 0.45]$

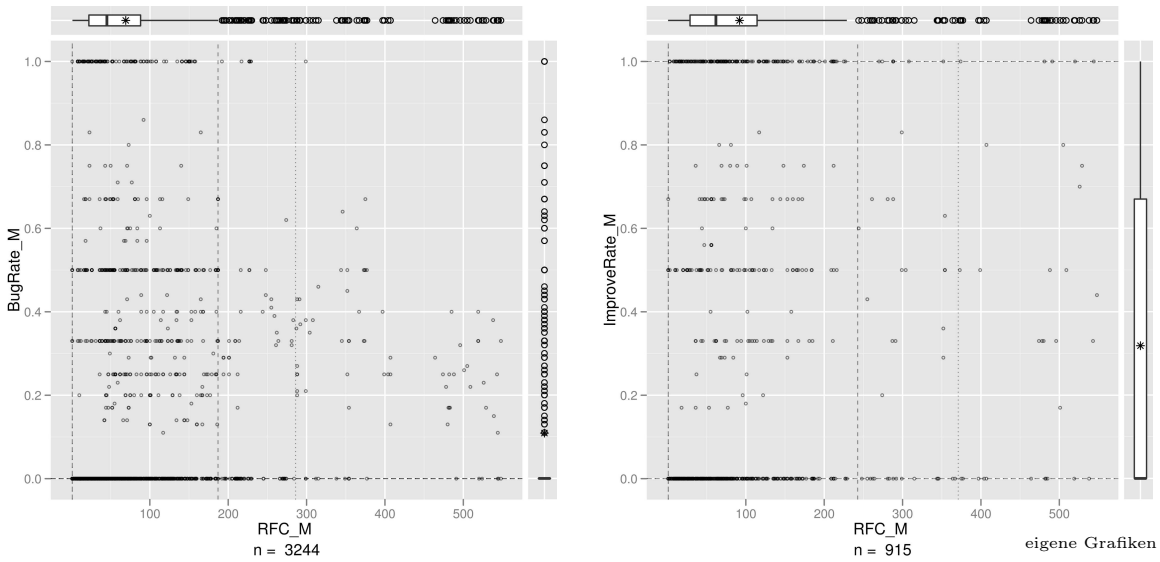


Abbildung 4.11 – Streudiagramme für $\mathrm{RFC_M}$ und $\mathrm{BUGRATE_M}$ beziehungsweise $\mathrm{IMPROVERATE_M}$

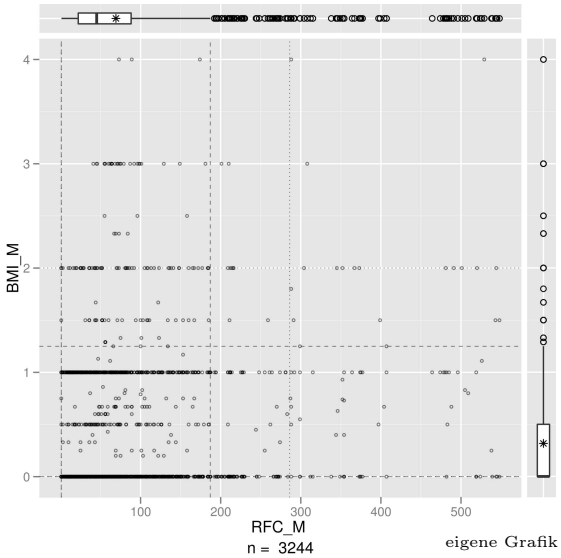


Abbildung 4.12 – Streudiagramm von $\mathrm{RFC_M}$ und $\mathrm{BMI_M}$

daher sachlich nicht mehr gerechtfertigt ist, wurde bereits oben erläutert.

Zusammenfassung Für Hypothese AP3 kann ebenfalls kein Gesamturteil abgegeben werden, da die Einzelergebnisse unter den Grundannahmen (siehe Abschnitt 4.3.1.1, S. 76) widersprüchlich sind beziehungsweise diesen widersprechen.

Festzuhalten ist, dass $\mathrm{RFC_M}$ moderat positiv ($r = 0,39$) mit der Anzahl der FEH-

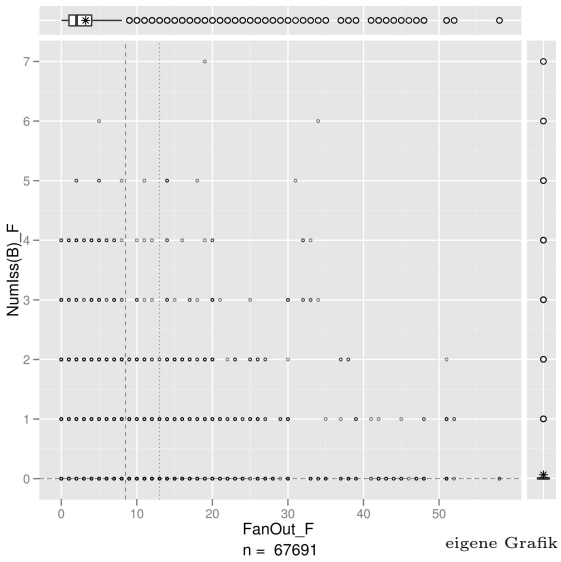


Abbildung 4.13 – Streudiagramm von FanOut_F und $\text{NumIss(B)}_\text{F}$

ler und, wider Erwarten, der monatlichen Lösungsrate von Fehlern ($r = 0,36$) korreliert. Schwache Rangkorrelationen bestehen mit dem Anteil der Fehler an allen Fällen ($R = 0,29$) und dem Verbesserungs-Anteil an gelösten Fällen ($R = 0,25$).

4.3.6.4 Hypothese AP4

Entgegen der von Ryder [80:148f.] berichteten, mit $r \approx 0,5$ ($p \leq 0,05$) moderaten Korrelation des ·Ausgangsgrades von Funktionen mit der Anzahl von Fehlerkorrekturen zeigte FanOut_F nur einen vernachlässigbaren linearen Zusammenhang mit der Fehler-Anzahl: $r = 0,16$, bei einem Konfidenzintervall $[0.15, 0.17]$). Abbildung 4.13, S. 96 scheint für die Ausreißer (nach Boxplot-Konvention) einen ***negativen*** Zusammenhang zu zeigen. Falls dieser existiert, wird er insgesamt durch die überwältigende Konzentration von Funktionen ohne Fehler (ersichtlich an der auf Null zusammengestauchten Box des Randverteilungsboxplots für $\text{NumIss(B)}_\text{F}$) überlagert. Die Hypothese AP4 muss verworfen werden.

4.3.6.5 Hypothese AP5

Das Maß für die ·Instabilität von Modulen, mit den beiden Varianten InstF_M und InstM_M, weist insgesamt praktisch keine Korrelation mit der durchschnittlichen

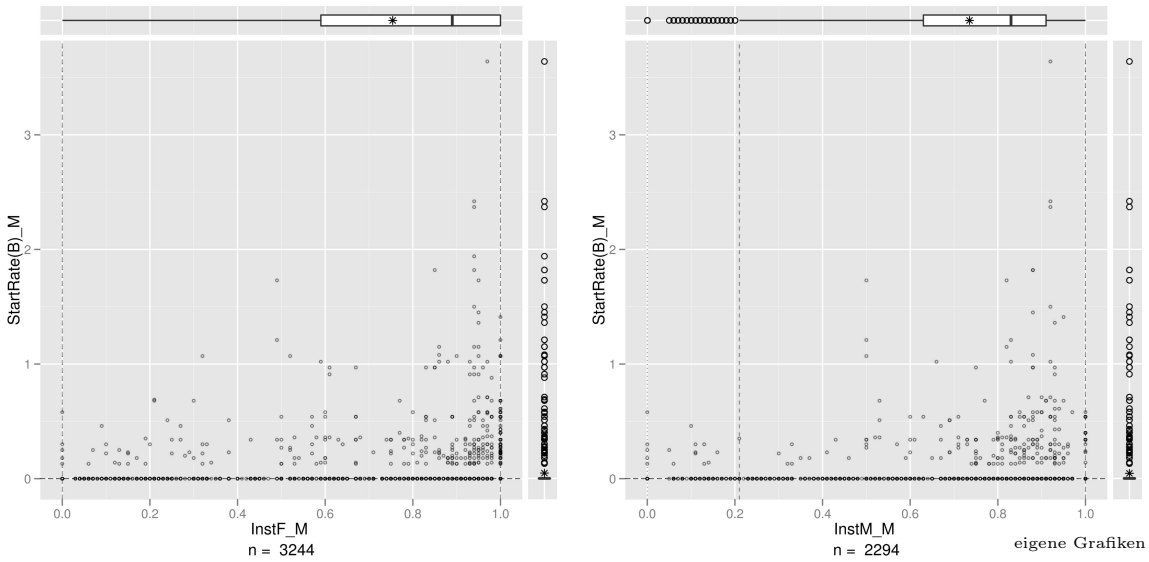


Abbildung 4.14 – Streudiagramme von INSTF_M beziehungsweise INSTM_M mit $\text{STARTRATE(B)}_\text{M}$

Anzahl neuer FEHLER pro Monat über die Lebenszeit der Module, $\text{STARTRATE(B)}_\text{M}$, auf: jeweils $r = 0,07$. In Abb. 4.14 ist jedoch zu erkennen, dass der Zusammenhang für Module, deren $\text{STARTRATE(B)}_\text{M}$-Wert nicht Null ist, deutlich stärker zu sein scheint. An den auf Null zusammengestauchten Boxen der Randverteilungen ist erneut der überwältigende Einfluss der großen Mehrheit FEHLERfreier Module zu erkennen, für die $\text{STARTRATE(B)}_\text{M}$ Null ist. Ob eine vollständigere Zuordnung von FEHLERN zu Programmkomponenten diesen Einfluss erheblich senken würde, wäre in weiteren Studien festzustellen.

Hypothese AP6 Hopkins und Hatton [44:8] (siehe Abschnitt 5.3, S. 105) berichten von einer schwachen, aber signifikanten und im Widerspruch zu einschlägigen Programmierrichtlinien ***negativen*** Korrelation der maximalen Verschachtelungstiefe von `if`-Verzweigungen mit der Anzahl an Fehlern in den entsprechenden Funktionen einer großen FORTRAN-Bibliothek. Abb. 5.1, S. 106 zeigt das Diagramm, das den von ihnen festgestellten negativen Zusammenhang auf Funktionsebene darstellt. Entgegen diesen Ergebnissen (und in Einklang mit den erwähnten Programmierrichtlinien) ergab sich in der vorliegenden Untersuchung auf Funktionsebene kein negativer, sondern ein positiver Zusammenhang, der allerdings nicht die geforderte Effektstärke erreicht.[101] Die Nullhypothese wird dennoch bei-

[101] Sowohl für MAXCASE_F als auch für MAXNEST_F: $r = 0,13$. Siehe auch Abb. D.3, S. 143.

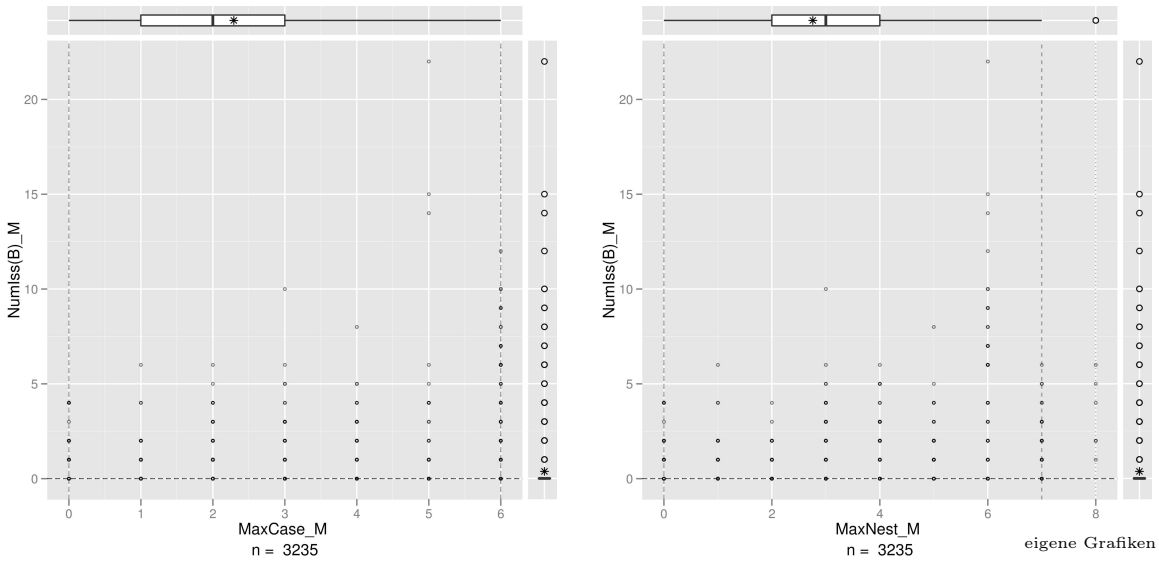


Abbildung 4.15 – Streudiagramme für MaxCase$_M$ und NumIss(B)$_M$ sowie MaxNest$_M$ und NumIss(B)$_M$

behalten, da Hypothese AP6 nicht bestätigt wurde.

Auf Modulebene jedoch konnte eine moderate ***positive*** Korrelation sowohl der maximalen Verschachtelungstiefe von `case`-Ausdrücken allein (MaxCase$_M$) als auch der maximalen Verschachtelungstiefe von `begin/end-`, `case-`, `fun-`, `if-` und `receive`-Ausdrücken (MaxNest$_M$) festgestellt werden: $r = 0,29$ beziehungsweise $r = 0,27$.[102] Dieser positive Zusammenhang auf Modulebene ist in Abbildung 4.15, S. 98 deutlich zu erkennen.

4.3.6.6 Hypothese T1

Für die beiden Unterhypothesen wurde die maximale Verschachtelungstiefe von `case`-Ausdrücken allein (MaxCase$_{FM}$) und von `begin`/`end-`, `case-`, `fun-`, `if-` und `receive`-Ausdrücken (MaxNest$_{FM}$) betrachtet, jeweils für die Funktions- und die Modulebene (erkennbar am Index F und M). Tabelle 4.7 auf der nächsten Seite zeigt die entsprechenden Kontingenztafeln.

Als Beträge der Φ-Koeffizienten ergeben sich (in der Lesereihenfolge der Tafeln) die niedrigen Werte $0,1$, $0,25$, $0,1$ und $0,23$, jeweils mit p und $\beta = 0,00$. Der

[102] Konfidenzintervalle: $[0.26, 0.32]$ beziehungsweise $[0.24, 0.30]$

Tabelle 4.7 – Kontingenztafeln für $\text{MaxCase}_{\text{FM}}$ beziehungsweise $\text{MaxNest}_{\text{FM}}$ und $\text{NumIss(B)}_{\text{FM}}$

		$\text{MaxCase}_{\text{F}}$	
		≤ 2	> 2
$\text{NumIss(B)}_{\text{F}}$	$= 0$	64979	3259
	> 0	2885	510

		$\text{MaxCase}_{\text{M}}$	
		≤ 2	> 2
$\text{NumIss(B)}_{\text{M}}$	$= 0$	1758	961
	> 0	225	450

		$\text{MaxNest}_{\text{F}}$	
		≤ 2	> 2
$\text{NumIss(B)}_{\text{F}}$	$= 0$	61848	6390
	> 0	2592	803

		$\text{MaxNest}_{\text{M}}$	
		≤ 2	> 2
$\text{NumIss(B)}_{\text{M}}$	$= 0$	1422	1297
	> 0	157	518

		LOC_{M}	
		≤ 400	> 400
$\text{NumIss(B)}_{\text{M}}$	$= 0$	2164	561
	> 0	325	353

Tabelle 4.8 – Kontingenztafel für LOC_{M} und $\text{NumIss(B)}_{\text{M}}$

Zusammenhang auf Moduleebene scheint stärker ausgeprägt zu sein als auf Funktionsebene. Ein möglicher Grund dafür ist, dass die Zuordnung von Fällen zu Programmkomponenten auf Modulebene vollständiger ist und damit das Übergewicht Fehler-freier Komponenten sinkt. Die Nullhypothese der Unabhängigkeit der Merkmale wird sowohl für $\text{MaxCase}_{\text{FM}}$ als auch für $\text{MaxNest}_{\text{FM}}$ (also ebenso auf Funktion- wie auf Modulebene) zugunsten von Hypothese T1 verworfen.

4.3.6.7 Hypothese T2

Mit der Kontingenztafel in Abschnitt 4.3.6.7 ergibt sich $\bar{\Phi} = 0.28$ mit p und $\beta = 0.00$. Somit wird die Nullhypothese zugunsten von Hypothese T2 verworfen, obzwar der Zusammenhang schwach ist.

4.3.6.8 Hypothese T3

Da der Schwellwert bei dieser Hypothese als Intervall angegeben ist, werden die Unter- und Obergrenze separat überprüft. Für die Kontingenztafeln in Tabelle 4.9 auf der nächsten Seite ist $\bar{\Phi} = 0.21$ beziehungsweise $\bar{\Phi} = 0.29$, jeweils mit p und $\beta = 0.00$. Die Nullhypothese wird sowohl für die Ober- als auch für die Untergrenze zugunsten von Hypothese T3 verworfen.

Tabelle 4.9 – Kontingenztafeln für NumFun_M und $\text{NumIss(B)}_\text{M}$

		NumFun_M	
		≤ 30	> 30
$\text{NumIss(B)}_\text{M}$	$= 0$	2215	510
	> 0	400	278

		NumFun_M	
		≤ 40	> 40
$\text{NumIss(B)}_\text{M}$	$= 0$	2527	198
	> 0	471	207

Tabelle 4.10 – Kontingenztafeln für LOC_F und $\text{NumIss(B)}_\text{F}$

		LOC_F	
		≤ 15	> 15
$\text{NumIss(B)}_\text{F}$	$= 0$	49137	19101
	> 0	1615	1780

		LOC_F	
		≤ 20	> 20
$\text{NumIss(B)}_\text{F}$	$= 0$	54578	13660
	> 0	1916	1479

4.3.6.9 Hypothese T4

Auch hier werden Unter- und Obergrenze des Schwellwertintervalls separat betrachtet (siehe Tabelle 4.10). Der Zusammenhang ist signifikant (jeweils p und $\beta = 0.00$), wenn auch mit $\bar{\Phi} = 0.11$ und $\bar{\Phi} = 0.12$ relativ schwach. Die Nullhypothese muss verworfen werden.

4.3.6.10 Hypothese T5

Zwei Varianten eines Schwellwertes für das Maß AvgLOC_M werden darauf untersucht, ob die entsprechenden Module sich in ihrem Fehlergehalt deutlich unterscheiden (siehe Tabelle 4.11). Der hypothetische Schwellwert auf Grundlage des Medians und der mittleren Abweichung vom Median (Hypothese T5.1) liegt bei $1,5 \cdot (\tilde{x}_{0,5} + \tilde{d}_{0,5}) = 28,65$, der auf dem arithemetischen Mittelwert und der Standardabweichung basierende liegt bei $1,5 \cdot (\bar{x} + s) = 37,5$ (Hypothese T5.2). Im ersten Fall ist das Ergebnis knapp ($\bar{\Phi} = 0.03$, $p = 0.06$, $\beta = 0.00$), im zweiten Fall deutlich nicht signifikant ($p = 0.81$, $\bar{\Phi} = 0.00$, $\beta = 0.00$), so dass die Nullhypothese in beiden Fällen zuungunsten Hypothese T5.2 beibehalten wird.

Tabelle 4.11 – Kontingenztafeln für AvgLOC_M und $\text{NumIss(B)}_\text{M}$

		AvgLOC_M	
		$\leq 28,65$	$> 28,65$
$\text{NumIss(B)}_\text{M}$	$= 0$	2543	176
	> 0	617	58

		AvgLOC_M	
		$\leq 37,5$	$> 37,5$
$\text{NumIss(B)}_\text{M}$	$= 0$	2622	97
	> 0	649	26

Tabelle 4.12 – Kontingenztafeln für AvgCalls$_M$ und NumIss(B)$_M$

		AvgCalls$_M$	
		$\leq 6,3$	$> 6,3$
NumIss(B)$_M$	$= 0$	2522	203
	> 0	610	68

		AvgCalls$_M$	
		$\leq 8,77$	$> 8,77$
NumIss(B)$_M$	$= 0$	2614	111
	> 0	641	37

Hypothese T6 Auch hier wurde sowohl der auf dem Median basierende Schwellwert $(6,3)$ als auch der nach Marinescu und Lanza [60:29] $(8,775)$ betrachtet. Tabelle 4.12 zeigt die entsprechenden Kontingenztafeln. Es existiert ein signifikanter, aber schwacher Zusammenhang mit dem Median-basierten Schwellwert ($\bar{\Phi} = 0.04$, $p = 0.03$, $\beta = 0.00$), so dass die Nullhypothese zugunsten Hypothese T6.1 verworfen werden muss. Das Ergebnis für den anderen Schwellwert ist nicht signifikant ($\bar{\Phi} = 0.03$, $p = 0.14$, $\beta = 0.00$), die Nullhypothese wird folglich zuungunsten Hypothese T6.2 beibehalten.

5 Ergebnisse und Diskussion

In diesem Kapitel werden die Ergebnisse der vorangehenden Untersuchung zusammengefasst und diskutiert.

5.1 Eignung der externen Maße zur Einschätzung von ·Wartbarkeit

Einige der zuvor getroffenen Grundannahmen, nach denen bestimmte externe Maße als Indikatoren für bessere oder schlechtere ·Wartbarkeit angenommen wurden, wurden durch die vorliegende Studie in Zweifel gezogen. Von der durchschnittlichen Anzahl pro Monat gelöster Fehler ($\text{EndRate(B)}_\text{M}$) wurde als Zeichen dafür angesehen, dass Fehler leicht behoben werden können. Es stellte jedoch heraus, dass diese Lösungsrate stark mit der Gesamtzahl von Fehlern zusammenhängt. Dieses Ergebnis deutet darauf hin, dass $\text{EndRate(B)}_\text{M}$ in der hier verwendeten Form nicht als Maß für ·Wartbarkeit gelten kann.

Als Indikator guter Wartbarkeit zumindest nicht widerlegt wurde der Anteil der Verbesserungen an allen gelösten Fällen ($\text{ImproveRate}_\text{M}$), der nur schwach positiv mit der Anzahl der Fehler zusammenhing.

Die mittlere Bearbeitungszeit von Fehlern ($\text{MedITT(B)}_\text{M}$), von der erwartet wurde, für schlechter wartbare Komponenten größere Werte anzunehmen, weist überraschenderweise einen erheblichen negativen Zusammenhang mit der Fehler-Anzahl auf. Wie oben erwähnt, liegt dies vermutlich nicht hauptsächlich in internen Qualitätsmerkmalen begründet, sondern etwa in höherer Priorisierung Fehler-haltiger Komponenten bei der Wartung.

5.2 Zusammenhänge zwischen internen und externen Qualitätsmerkmalen

Die Ergebnisse über lineare Zusammenhänge von internen und externen Maßen sind uneinheitlich. Keine der Haupthypothesen, die sich auf die Validierungskriterien ·Assoziation und ·Parallele Veränderung beziehen, konnte vollständig angenommen oder verworfen werden. Die schwächeren Hypothesen zum Kriterium ·Trennschärfe konnten überwiegend bestätigt werden.

5.2.1 Assoziation und parallele Veränderung

Die summierte zyklomatische Komplexität der Funktionen eines Moduls und die Mächtigkeit der ·Antwortmenge, das heißt der potentiell durch einen Funktionsaufruf ausgelösten weiteren Funktionsaufrufe ($\mathrm{WMC(CYC)_M}$ beziehungsweise $\mathrm{RFC_M}$), wiesen den stärksten Zusammenhang mit der Anzahl der FEHLER,[103] der monatlichen Durchschnittsrate behobener FEHLER,[104] dem Anteil der FEHLER an allen Fällen eines Moduls[105] mit dem Anteil von VERBESSERUNGEN an den gelösten Fällen im Fallbearbeitungssystem.[106] Der starke lineare Zusammenhang der beiden Maße könnte auf einen gemeinsamen Hintergrundeinfluss hindeuten, der die Ähnlichkeit der Ergebnisse erklärt. Dies konnte in dieser Untersuchung nicht überprüft werden.

Die Stärke der ·Kopplung eines Moduls in Bezug auf andere Module wies keinen erheblichen Zusammenhang mit irgendeinem der untersuchten Maße für ·Wartbarkeit auf. Dies steht im Gegensatz zu den üblichen Empfehlungen, wonach starke Kopplung von Modulen einen negativen Einfluss auf ihre Handhabbarkeit haben sollte [vgl. 5:474]. Ebenso konnte nicht bestätigt werden, dass die ·Instabilität von Modulen [61:262] mit der Häufigkeit von Änderungen, gemessen am Aufkommen neuer ·Fälle im Fallbearbeitungssystem, korrespondiert. Ryders Beobachtung eines starken Zusammenhangs zwischen dem Ausgangsgrad von Funktionen, das heißt dem Ausmaß, in dem sie andere Funktionen aufrufen, und dem Fehlergehalt von Funktionen [80:148] konnte nicht bestätigt werden.

[103] $r = 0,38$ beziehungsweise $r = 0,39$
[104] $r = 0,34$ beziehunsgweise $r = 0,36$
[105] Jeweils $R = 0,29$
[106] $R = 0,22$ beziehunsgweise $R = 0,25$

Auch das Ergebnis von Hopkins und Hatton [44:8], dass tief verschachtelte Fallunterscheidungen im Widerspruch zu gängigen Empfehlungen mit ***weniger*** Fehlern einhergehen, konnte nicht reproduziert werden. Ohnehin hatten die Autoren dies auf externe Einflüsse wie größere Sorgfalt der Entwickler zurückgeführt, die durch interne Softwaremaße offensichtlich nicht erfasst werden.

5.2.2 Trennschärfe

Es existiert ein schwacher Zusammenhang, demzufolge für Funktionen und Module, in denen `begin/end`-, `fun`-, `if`- oder `receive`-Ausdrücke in drei oder mehr Ebenen verschachtelt sind, deutlich mehr Fehlerberichte erstellt werden, während die Verschachtelungstiefe von `case`-Ausdrücken keinen erheblichen Zusammenhang aufweist.

Etwas stärker ist der Unterschied bezüglich der Zeilenanzahl und der Anzahl der Funktionen von Modulen: Module, die mehr als 400 Zeilen oder mehr als 30 oder 40 Funktionen enthalten, haben deutlich mehr Fehlerberichte als solche, die kürzer sind oder weniger Funktionen enthalten.

Keine nennenswerten Unterschiede in der FEHLER-Anzahl konnten bei einem Median-basierten Schwellwert von 15 bis 20 Zeilen für einzelne Funktionen oder einem auf dem arithmetischen Mittel basierenden Schwellwert von 28 bis 38 Zeilen im Durchschnitt pro Funktion eines Moduls festgestellt werden. Allerdings war das Ergebnis für den Median-basierten Schwellwert nur knapp insignifikant, während sich der zweite Schwellwert als völlig unbrauchbar erwies.

In Bezug auf die durchschnittliche Anzahl verschiedener Funktionsaufrufe pro Funktion eines Moduls konnte ein signifikanter Unterschied zwischen Modulen diesseits und jenseits eines Median-basierten Schwellwertes festgestellt werden: Module mit durchschnittlich mehr als $6,3$ Aufrufen hatten deutlich mehr FEHLER als Module mit weniger Aufrufen.

5.2.3 Vergleich einiger Ergebnisse mit $\mathsf{LOC_{FM}}$

Ein Vergleich der Pearson-Korrelationskoeffizienten für den Zusammenhang der Zeilenanzahl $\mathrm{LOC_M}$ mit den in den Hypothesen AP1 bis AP3 untersuchten ex-

Tabelle 5.1 – Linearer Zusammenhang der Zeilenanzahl LOC_M mit den untersuchten externen Maßen. Das Konfidenzintervall ist bezüglich BMI_M mit $[0.16, 0.22]$ und bezüglich BugRate_M mit $[0.14, 0.21]$ nicht ausreichend für die Einstufung als erheblicher Zusammenhang.

$\text{NumIss(B)}_\text{M}$	BugRate_M	$\text{MedITT(B)}_\text{M}$	BMI_M	$\text{EndRate(B)}_\text{M}$	$\text{ImproveRate}_\text{M}$
0,42	0,17	0,00	0,19	0,36	0,11

ternen Maßen zeigt, dass die Maße WMC(Cyc)_M und RFC_M bei Betrachtung lediglich des linearen Zusammenhangs keine ·Verbesserungsvalidität gegenüber der Zeilenzahl besitzen. Deren Zusammenhang mit den Maßen $\text{NumIss(B)}_\text{M}$ und $\text{EndRate(B)}_\text{M}$, auf die bezogen die Hypothesen AP1.1 und AP1.5 beziehungsweise AP3.1 und AP3.4 bestätigt wurden, ist ebenso stark wie oder stärker als bei WMC(Cyc)_M und RFC_M. Es wäre zu prüfen, ob sich bei Betrachtung von Untergruppen der Stichproben oder von nichtlinearen Zusammenhängen bedeutende Unterschiede der drei Maße zeigen. Bis dahin ist der Zusatznutzen von WMC(Cyc)_M und RFC_M zweifelhaft.

5.3 Vergleich mit der Untersuchung von Hopkins und Hatton

Ähnlich wie in der vorliegenden Studie fanden Hopkins und Hatton [44] bei einer Fallstudie mit einer großen, ausgereiften[107] Fortran-Bibliothek bei 15 bekannten Software-Maßen zwar signifikante, aber schwache Korrelationen mit der Fehlerzahl (siehe beispielsweise Abb. 5.1 auf der nächsten Seite). Wie in Abschnitt 2.3.4.2, S. 40 diskutiert, kann dies zum Teil darauf zurückgeführt werden, dass einzelne Komponenten verschiedene Nutzungsprofile aufweisen und die Anzahl der bekannten und gemeldeten Fehler von diesen Nutzungsunterschieden verzerrt wird. Um dies auszugleichen, gruppieren die Autoren die Messwerte der internen Maße nach ihrer Fehleranzahl und untersuchen den Zusammenhang der Fehleranzahl mit den arithmetischen Mittelwerten der Gruppen [44:12]. So erhalten sie deutlich stärkere Korrelationen in der vermuteten Richtung, durch die Datenreduktion allerdings auch mehr insignifikante Ergebnisse.

[107] 260000 Zeilen, 3600 Funktionen, 20 Jahre Nutzung.

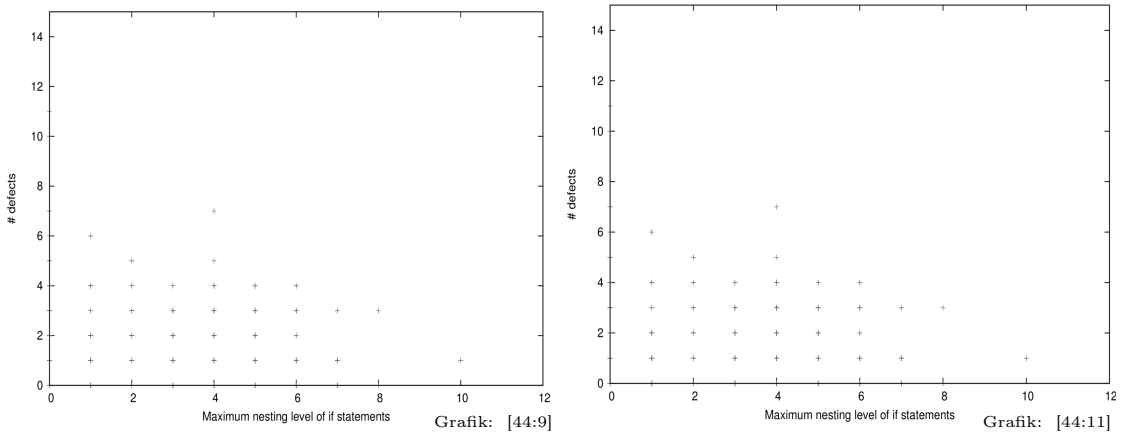


Abbildung 5.1 – Streudiagramme der Untersuchung von Hopkins und Hatton [44]. Im linken Diagramm ist die Anzahl ausführbarer Zeilen mit der zugehörigen Fehleranzahl dargestellt, im rechten Diagramm die maximale Verschachtelungstiefe von `if`-Anweisungen.

Dieser Ansatz konnte für diese Studie nicht mehr vollständig durchgeführt werden. Beispielhaft sei aber auf Abb. 5.2 auf der nächsten Seite verwiesen, wo die nach dem Wert von $\text{StartRate}_\text{M}$ gruppierten Module und die dazugehörigen Medianwerte des ·Instabilitäts-Maßes InstM_M dargestellt sind. Die lineare Korrelation nach Pearson beträgt $r = 0,32$, allerdings mit einem großen Konfidenzintervall von $[0.03, 0.55]$. Angesichts der reduzierten Stichprobengrößen nicht überraschend waren unter einigen so ermittelten Korrelationskoeffizienten deutlich mehr insignifikante Ergebnisse als ohne diese Transformation der Daten. Weitere Untersuchungen müssten den Nutzen dieses Verfahrens klären.

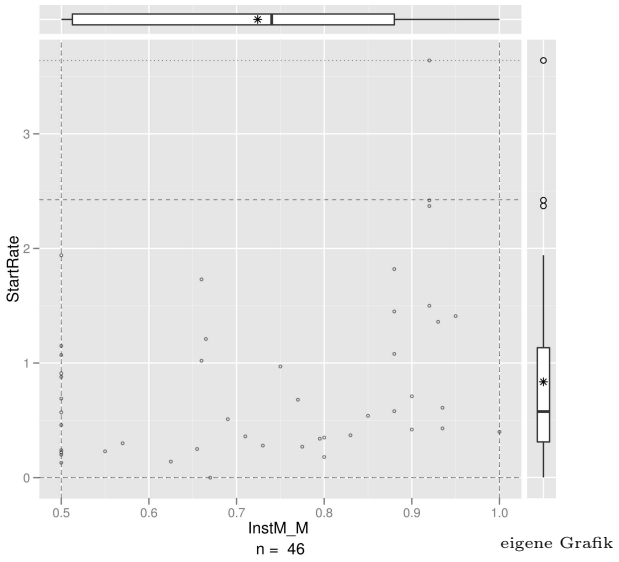


Abbildung 5.2 – Streudiagramm von StartRate$_M$ und dem Median der InstM$_M$-Werte aller Module eines StartRate$_M$-Wertes

6 Zusammenfassung

Mit dieser Arbeit liegt erstmals in der veröffentlichten deutsch- und englischsprachigen Literatur eine empirische Studie zur Validierung von Softwaremaßen für eine funktionale Programmiersprache, ERLANG, vor, die ein großes, professionell eingesetztes Softwareprodukt als Untersuchungsobjekt hat. Die Untersuchung konnte zu einer Reihe von internen Maßen signifikante Zusammenhänge mit externen Qualitätsmerkmalen der untersuchten Software, EJABBERD, nachweisen. Eine Reihe von Programmierrichtlinien für ERLANG konnte nach Kenntnis des Autors erstmals empirisch belegt werden.

Für die Erhebung der internen und externen Messwerte wurde eine Messumgebung entwickelt, die für weitere empirische Untersuchungen von ERLANG-Systemen genutzt und weiterentwickelt werden kann. Die Rohdaten, die im Rahmen dieser Arbeit nur zu einem kleinen Teil ausgewertet werden konnten, können für weitere Untersuchungen bereitgestellt werden und würden zentrale Datenbanken wie FLOSSMetrics[108] oder PROMISE[109] bereichern, die bisher kaum Messwerte funktionaler Programme enthalten.

Nicht zuletzt, weil aus Mangel an Alternativen für die statische Analyse des Untersuchungsobjekts auf ein Messwerkzeug (REFACTORERL) zurückgegriffen werden musste, das sich noch im Prototypen-Stadium befindet, hat die Untersuchung den Charakter einer Pilot-Studie. Im Folgenden werden kurz einige Aspekte diskutiert, in denen die Studie verbesserungsbedürftig erscheint, sowie ein Ausblick auf mögliche Erweiterungen gegeben.

[108] http://flossmetrics.org/sections/deliverables/WP1
[109] http://promisedata.org/

7 Offene Fragen & Ausblick

Abschließend werden nun notwendige Verbesserungen festgehalten und mögliche Erweiterungen vorgeschlagen.

7.1 Verbesserungen der Untersuchung

7.1.1 Erfassung des Lebenszyklus von Fällen

Fälle durchlaufen im Fallbearbeitungssystem verschiedene Bearbeitungsphasen (siehe Abschnitt 2.2.4, S. 31), die in einem Änderungsprotokoll festgehalten werden. Die Daten über die zeitliche Abfolge dieser Phasen werden vom entwickelten Fall-Extraktionsprogramm derzeit nicht erfasst, weil sie nicht wie die statischen Falldaten als XML-Dateien bereitstehen, sondern aus semistrukturierten HTML-Dateien extrahiert werden müssen, wofür der Bearbeitungsrahmen der Arbeit nicht ausreichte.

7.1.2 Präzisere Erfassung der Bearbeitungszeit von Fällen

Die Erfassung der dynamischen Veränderung eines Falls würde es erlauben, die Bearbeitungszeit von Fällen, aus der Rückschlüsse auf die Effizienz der Problembehebung im untersuchten Softwareprojekt gezogen werden können, präziser zu erfassen. Derzeit wird der für die Lösung eines Falls betriebene Aufwand grob überschätzt, da dafür die gesamte Zeitspanne von Erstellung bis Schließung eines Falls angesetzt wird. Dabei wird ignoriert, dass in der Bearbeitung von Fällen Unterbrechungen eintreten können und tatsächlich eintreten, in denen sie keinen Aufwand verursachen. Diese Unterbrechungen sind als Ereignisse im Änderungsprotokoll jedes Falles hinterlegt.

7.1.3 Vollständigere Verknüpfung von Fällen und Programmkomponenten

Bird u. a. [9:124] verwenden für ihre Verknüpfung von Fällen zu Programmkomponenten die BLAME-Funktion des Versionsverwaltungssystems GIT, um für jede Programmzeile festzustellen, mit welchem Commit, das heißt zu welchem Zeitpunkt sie ursprünglich ins System eingefügt wurde. Auf Grundlage dieser Information kann unter Umständen der Zuordnungszeitraum von Fällen zu Programmkomponenten auch zeitlich nach vorn erweitert werden, was zu einer vollständigeren Datengrundlage für die statistische Analyse führen würde.

7.1.4 Verfeinerung der statistischen Untersuchung

Die verwendeten statistischen Verfahren erfassen lediglich lineare Zusammenhänge zwischen den Messwerten. Bei der Überprüfung der Hypothesen (siehe Abschnitt 4.3.6, S. 87) wurden wiederholt Anzeichen nichtlinearer Zusammenhänge in den Streudiagrammen festgestellt. Diese könnten mit fortgeschritteneren statistischen Verfahren quantifiziert und auf ihre Signifikanz hin getestet werden.

Unabhängig davon sollten die Daten nach verschiedenen Einzelmerkmalen gruppiert untersucht werden, um Zusammenhänge aufdecken zu können, die durch die hier verfolgte Gesamtbetrachtung überdeckt blieben.

7.2 Mögliche Erweiterungen

7.2.1 Weitere externe Maße auf Grundlage von Fall-Daten und Änderungsprotokoll

Die im Fallbearbeitungssystem und im Änderungsprotokoll verfügbaren Daten wurden nur zu einem kleinen Teil genutzt. Weitere Untersuchungen könnten die anderen ·Fall-Typen, wie VERBESSERUNGEN oder AUFGABEN, einbeziehen. Darüber hinaus erlauben die Daten auch Rückschlüsse auf den „menschlichen Faktor“. Die Anzahl der Entwickler oder etwa ihre Auslastung, gemessen an der Zahl der zugewiesenen ·Fälle, oder Erfahrung, gemessen an der Zahl bisher gelöster Fälle,

könnten berücksichtigt werden, um ein besseres Verständnis vom Zusammenwirken der verschiedenen Faktoren zu erlangen.

7.2.2 Empirischer Vergleich von funktionaler und imperativer Programmierung

Funktionaler Programmierung wird nachgesagt, gewisse Vorteile gegenüber der imperativen Programmierung zu haben, u.a.: leichtere Verständlichkeit oder „Klarheit“ durch die an die Mathematik angelehnte Ausdrucksweise; leichtere Testbarkeit; größere Wiederverwendbarkeit etc. Bothe [12:16] sieht den Wert deskriptiver Programmiersprachen allgemein darin, dass sie das zu lösende Problem in den Mittelpunkt stellen, nicht den Weg zu seiner Lösung. Ryder [80:63] glaubt, dass funktionale Programme weniger unerwartetes oder nicht vorhersagbares Verhalten zeigen, da Ausdrücke genau eine Wirkung und keine Nebenwirkungen haben. Die behaupteten Vorteile der funktionalen Programmierung werden meist nicht mit empirischen Daten belegt (siehe Abschnitt 1.2, S. 7). Solide validierte Softwaremaße könnten als Grundlage dienen, um die Frage von Berg [6:3] zu beantworten: “[H]ow different is [functional programming] from the ‘classical’ imperative programming style? Is functional programming *good* for the development of software: can these programs be developed in a shorter time; are functional programs more reliable; are such programs easier to maintain?”

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[1] Douglas Adams. *The Ultimate Hitchhiker's Guide: Five Complete Novels and One Story*. Gramercy Books, 2005, S. 815 (siehe S. 32).

[2] Alfred V. Aho, Ravi Sethi und Jeffrey D. Ullmann. *Compilerbau*. Oldenbourg, 1999 (siehe S. 28–30).

[3] Herbert Amann und Joachim Escher. *Statistik III*. Birkhäuser, 2001 (siehe S. 22).

[4] Klaus Backhaus u. a. *Multivariate Analysemethoden*. 2006 (siehe S. 9).

[5] Helmut Balzert. *Lehrbuch der Software-Technik*. Spektrum Akademischer Verlag, 1998 (siehe S. 22, 24, 25, 52, 78, 103).

[6] Klaas G. van den Berg. „Software Measurement and Functional Programming“. Diss. 1995 (siehe S. 8–10, 27, 76, 111).

[7] Manuel F. Bertoa, Antonio Vallecillo und Félix García. „An Ontology for Software Measurement“. In: *Ontologies for Software Engineering and Software Technology*. Springer, 2006. Kap. 6, S. 175–196 (siehe S. 20, 21).

[8] Dennis Bijlsma. „Indicators of Issue Handling Efficiency“. Masterarbeit. 2010 (siehe S. 56, 57, 85).

[9] Christian Bird u. a. „Fair and Balanced? Bias in Bug-Fix Datasets“. In: *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM. 2009, S. 121–130 (siehe S. 41, 66, 110).

[10] Richard Bird und Philip Wadler. *Einführung in die funktionale Programmierung*. Hanser, 1992 (siehe S. 43).

[11] Jürgen Bortz und Christof Schuster. *Statistik für Human- und Sozialwissenschaftler*. Springer, 2010 (siehe S. 19, 20, 23).

[12] Klaus Bothe. „Von Algol nach Java: Kontinuität und Wandel von Programmiersprachen“. In: *Informatik : Aktuelle Themen im historischen Kontext*. Hrsg. von Wolfgang Reisig und Johann-Christoph Freytag. Springer-Verlag, 2006, S. 1–16 (siehe S. 42, 44, 111).

[13] Istvan Bozó u. a. „Using Impact Analysis Based Knowledge For Validating Refactoring Steps“. In: *Studia Universitatis Babes-Bolyai*. Informatica 56.3 (2011) (siehe S. 59).

[14] William J. Brown u. a. *AntiPatterns: refactoring software, architectures, and projects in crisis*. Wiley, 1998 (siehe S. 26).

[15] Francesco Cesarini und Simon Thompson. *Erlang Programming*. 2009 (siehe S. 75).

[16] Shyam R. Chidamber und Chris F. Kemerer. „A Metrics Suite for Object Oriented Design“. In: *IEEE Transactions on Software Engineering* 20.6 (Juni 1994), S. 476–493 (siehe S. 52, 53, 77, 78).

[17] Günter Clauß und Heinz Ebner. *Grundlagen der Statistik*. Berlin: Volk und Wissen, 1974 (siehe S. 82).

[18] Jacob Cohen. *Statistical power analysis for the behavioral sciences*. 2. Aufl. Lawrence Erlbaum Associates, 1988 (siehe S. 87).

[19] IEEE Computer Society. Standards Coordinating Committee und IEEE Standards Board. „IEEE Standard Glossary of Software Engineering Terminology“. In: *IEEE Standard Glossary of Software Engineering Terminology*. Hrsg. von Jane Radatz. Institute of Electrical und Electronics Engineers. New York, N.Y: Institute of Electrical und Electronics Engineers, 1990 (siehe S. 15).

[20] Ely Deckers. „Verifying properties of RefactorErl-transformations using QuickCheck“. URL: `http://www.ru.nl/publish/pages/578936/verifying_properties_of_refactorerl-transformations_-_ely_deckers.pdf`, abgerufen am 12.04.2012. Masterarbeit. 2010 (siehe S. 59).

[21] Jean-Marc Desharnais. *Analysis of ISO-IEC 9126 and 25010*. Vortragsfolien. URL: `http://www.cmpe.boun.edu.tr/courses/cmpe58V/fall2009/06a-Analysisof9126-2,3,4short.pdf`, abgerufen am 12.04.2012. 2009 (siehe S. 17).

[22] Edsger W. Dijkstra. „EWD268 : Structured Programming“. In: (Aug. 1969). URL:`http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD268.PDF`, abgerufen am 9. Mai 2012 (siehe S. 24).

[23] DIN. *Internationales Wörterbuch der Metrologie – International Vocabulary of Basic and General Terms in Metrology*. 2. Aufl. Beuth Verlag, 1994 (siehe S. 123–125).

[24] DIN. *Internationales Wörterbuch der Metrologie – International Vocabulary of Basic and General Terms in Metrology*. 3. Aufl. Beuth Verlag, 2010 (siehe S. 23, 123, 125).

[25] DIN. *Software-Ergonomie : Empfehlungen für die Programmierung und Auswahl von Software*. 1. Auflage (CD-Version). Bd. 354. DIN-Taschenbuch. Beuth Verlag, 2004 (siehe S. 16).

[26] Klas Eriksson, Mike Williams und Joe Armstrong. *Program Development Using Erlang - Programming Rules and Conventions*. Ericsson. März 1996 (siehe S. 30, 75, 79, 80).

[27] Norman E. Fenton und M. Neil. „A Critique of Software Defect Prediction Models“. In: *Software Engineering, IEEE Transactions on* 25.5 (1999), S. 675–689 (siehe S. 25).

[28] Norman E. Fenton und Shari Lawrence Pfleeger. *Software Metrics : A Rigorous & Practical Approach*. 2. Aufl. PWS Publishing Co., 1996 (siehe S. 16, 49).

[29] Norman E. Fenton, Robin W. Whitty und Agnes A. Kaposi. „A generalised mathematical theory of structured programming“. In: *Theoretical Computer Science* 36 (1985), S. 145–171 (siehe S. 30).

[30] Martin Fowler. *Refactoring oder: wie Sie das Design vorhandener Software verbessern*. dt. Ausg. Addison-Wesley, 2005 (siehe S. 25, 26).

[31] Erich Gamma u. a. *Entwurfsmuster : Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, 2001 (siehe S. 26).

[32] Félix García u. a. „Towards a Consistent Terminology for Software Measurement“. In: *Information & Software Technology* (2006), S. 631–644 (siehe S. 20, 23, 123–125).

[33] Walter Gellert u. a. *Mathematik*. Kleine Enzyklopädie. Leipzig: VEB Bibliographisches Institut, 1965 (siehe S. 22).

[34] Gerd Gigerenzer. *Messung und Modellbildung in der Psychologie*. Ernst Reinhardt GmbH & Co., 1981 (siehe S. 28).

[35] Stephen Jay Gould. *Der falsch vermessene Mensch*. Suhrkamp Verlag, 1983, S. 400 (siehe S. 18).

[36] Todd L. Graves u. a. „Predicting fault incidence using software change history“. In: *IEEE Transactions on Software Engineering* 26.7 (2000), S. 653–661 (siehe S. 55, 69, 83).

[37] Jonathan L. Gross und Jay Yellen. *Handbook of Graph Theory*. Discrete mathematics and its applications. CRC, 2004 (siehe S. 51).

[38] Rudolf Halin. *Graphentheorie*. Akademie-Verlag Berlin (Lizenzausgabe), 1989 (siehe S. 51).

[39] T. Hall u. a. „A Systematic Literature Review on Fault Prediction Performance in Software Engineering“. In: () (siehe S. 25).

[40] Maurice H. Halstead. *Elements of Software Science*. Operating and Programming Systems Series. Elsevier, 1977 (siehe S. 8).

[41] Matthew S. Hecht. *Flow analysis of computer programs*. Programming languages series ; [5]. Elsevier Science Inc., 1977 (siehe S. 29, 30).

[42] Ilja Heitlager, Tobias Kuipers und Joost Visser. „A practical model for measuring maintainability“. In: *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*. IEEE. 2007, S. 30–39 (siehe S. 16, 18).

[43] Pieter Hooimeijer und Westley Weimer. „Modeling Bug Report Quality“. In: (2007), S. 34–43 (siehe S. 41).

[44] Tim Hopkins und Les Hatton. „Exploring defect correlations in a major Fortran numerical library“. URL: `http://www.leshatton.org/Documents/NAG01_01-08.pdf`, abgerufen am 12.04.2012. 2008 (siehe S. 27, 40, 42, 75, 79, 83, 89, 97, 104–106).

[45] James W. Howatt und Albert L. Baker. „Definition and Design of a Tool for Program Control Structure Measures“. In: *Proc. COMPSAC 85* 214 (1985), S. 214–220 (siehe S. 18).

[46] IEEE. *IEEE Standard for a Software Quality Metrics Methodology*. IEEE Standards Dept., 1998 (siehe S. 15, 24, 33, 86).

[47] ISO. *Information Technology : Software Product Evaluation : Quality Characteristics and Guidelines for their Use*. International Standards Organisation, 1991 (siehe S. 16).

[48] ISO. *Software engineering : Software product Quality Requirements and Evaluation (SQuaRE) : Measurement reference model and guide*. International Standards Organisation, 2007 (siehe S. 33).

[49] ISO. *Systems and software engineering : Systems and software Quality Requirements and Evaluation (SQuaRE) : System and software quality models*. International Standards Organisation, 2011 (siehe S. 16, 17, 32).

[50] JIRA. *JIRA Documentation 4.4.x*. URL: `http://confluence.atlassian.com/download/attachments/71598773/JIRA+4.4+Documentation+%28PDF%29+20110819.pdf`, abgerufen am 04.03.2012. JIRA. 2011 (siehe S. 32).

[51] Elroy Jumpertz. „Using QuickCheck and Semantic Analysis to Verify Correctness of Erlang Refactoring Transformations“. URL: `http://www.ru.nl/publish/pages/578936/refactoring_transformations_elroy_jumpertz.pdf`, abgerufen am 12.04.2012. Masterarbeit. 2010 (siehe S. 59).

[52] Stephen H. Kan. *Metrics and Models for Software Quality Engineering*. Pearson Education Inc., 2003 (siehe S. 57).

[53] Roland Király und Róbert Kitlei. „Application of Complexity Metrics in Functional Languages“. In: *Proceedings of the 8th Joint Conference on Mathematics and Computer Science, Selected Papers*. URL: `http://www.selyeuni.sk/macs/pdf/MaCS-2010.pdf`, abgerufen am 12.04.2012. 2010, S. 267–282 (siehe S. 8, 12, 58).

[54] Barbara A. Kitchenham, T. Dyba und M. Jorgensen. „Evidence-based software engineering“. In: *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*. IEEE. 2004, S. 273–281 (siehe S. 33).

[55] David H. Krantz u. a. *Foundations of Measurement : Additive and polynomial representation*. Bd. 1. 1991 (siehe S. 22, 23).

[56] Ulf Leser und Felix Naumann. *Informationsintegration*. dpunkt-Verl., 2006 (siehe S. 20, 65).

[57] Peter Liggesmeyer. *Software-Qualität : Testen, Analysieren und Verifizieren von Software*. Spektrum Akademischer Verlag, 2002 (siehe S. 15, 22, 25, 26, 42, 52).

[58] Wolfram-Manfred Lippe. *Funktionale und Applikative Programmierung*. Springer, 2009 (siehe S. 44, 45).

[59] Bart J. H. Luijten. „The Influence of Software Maintainability on Issue Handling“. Masterarbeit. 2010 (siehe S. 55, 56, 65, 70).

[60] Radu Marinescu und Michele Lanza. *Object-oriented metrics in practice*. New York, 2006 (siehe S. 23–27, 54, 80, 101).

[61] Robert C. Martin. *Agile Software Development : Principles, Patterns, and Practices*. Pearson Education Inc., 2003 (siehe S. 50, 51, 53, 103).

[62] Thomas J. McCabe. „A Complexity Measure“. In: *IEEE Transactions on Software Engineering*. SE 2.4 (Dez. 1976), S. 308–320 (siehe S. 52).

[63] Reginald N. Meeson Jr. „Functional Programming“. In: *Encyclopedia of Software Engineering* 1 (1994). Hrsg. von John J. Marciniak, S. 524–526 (siehe S. 42, 43).

[64] Beate Meffert und Olaf Hochmuth. *Werkzeuge der Signalverarbeitung*. Pearson Studium, 2004 (siehe S. 82, 134, 135, 138, 139).

[65] Andrew Meneely, Ben Smith und Laurie Williams. *Software Metrics Validation Criteria: A Systematic Literature Review*. Techn. Ber. 2010 (siehe S. 33–37, 130–133).

[66] Meine J.P. van der Meulen und Miguel A. Revilla. „Correlations between Internal Software Metrics and Software Dependability in a Large Population of Small C/C++ Programs“. In: ISSRE '07. 18th IEEE International Symposium on Software Reliability Engineering. IEEE Computer Society, 2007, S. 203–208 (siehe S. 81).

[67] Jan Midtgaard. „Control-flow analysis of functional programs“. In: (). URL: `cs.au.dk/~jmi/Midtgaard-CSur-final.pdf`, abgerufen am 26.04.2012 (siehe S. 58).

[68] Jack Moffit. *Professional XMPP*. Wiley, 2010 (siehe S. 62).

[69] Trevor T. Moores. „Applying complexity measures to rule-based prolog programs“. In: *Journal of Systems and Software* 44.1 (Dez. 1998), S. 45–52 (siehe S. 79).

[70] Harvey Motulsky. *Intuitive Biostatistics*. Oxford University Press, 1995 (siehe S. 87).

[71] Randall Munroe. *xkcd : volume 0*. Breadpig Inc, 2009 (siehe S. 75).

[72] Jerome L. Myers und Arnold Well. *Research design and statistical analysis*. 2. Aufl. Bd. 1. Lawrence Erlbaum, 2003 (siehe S. 9).

[73] Hartmut Noltemeier. *Graphentheorie : mit Algorithmen und Anwendungen*. Walter de Gruyter, 1976 (siehe S. 51).

[74] Frank R. Oppedijk. „Comparison of the SIG Maintainability Model and the Maintainability Index“. Masterarbeit. 2008 (siehe S. 18).

[75] Bernd Orth. „Grundlagen des Messens“. In: *Messen und Testen* 4 (1983), S. 136–180 (siehe S. 19, 23).

[76] Peter Pepper. *Funktionale Programmierung in OPAL, ML, HASKELL und GOFER*. 2. überarb. Aufl. Springer-Verlag, 2002 (siehe S. 42–44).

[77] Marian Petre und Russel L. Winder. „Programming Languages: Models and Programming Styles“. In: *Encyclopedia of Software Engineering* 2 (1994). Hrsg. von John J. Marciniak, S. 892–900 (siehe S. 42).

[78] Gustav Pomberger und Wolfgang Pree. *Software Engineering : Architektur-Design und Prozessorientierung*. 3. Aufl. Hanser, 2004 (siehe S. 50).

[79] *RefactorErl 0.9.12.01 Manual*. Eötvös Loránd University und Ericsson Hungary. 2012 (siehe S. 49–52, 59).

[80] Christopher Ryder. „Software Measurement for Functional Programming“. Diss. 2004 (siehe S. 7, 8, 10–12, 24, 26, 52, 66, 75, 76, 78, 81, 96, 103, 111).

[81] Christopher Ryder und Simon Thompson. „Software metrics: measuring haskell“. In: *Trends in Functional Programming* (2005), S. 31–46 (siehe S. 8, 10).

[82] Peter Saint-Andre, Kevin Smith und Remko Tronçon. *XMPP: The Definitive Guide*. O'Reilly, 2009 (siehe S. 62).

[83] Bernd-Holger Schlingloff. „Softwarequalität : Geschichte und Trends“. In: *Informatik : Aktuelle Themen im historischen Kontext*. Hrsg. von Wolfgang Reisig und Johann-Christoph Freytag. Springer-Verlag, 2006, S. 329–345 (siehe S. 23, 40).

[84] Norman F. Schneidewind. „Methodology for validating software metrics“. In: *Software Engineering, IEEE Transactions on* 18.5 (1992), S. 410–422 (siehe S. 24, 25).

[85] Uwe Schöning. *Theoretische Informatik : kurzgefasst*. Spektrum Akademischer Verlag, 2003 (siehe S. 52).

[86] Diomidis Spinellis. *Code Quality*. 1. Aufl. Addison-Wesley, 2006 (siehe S. 24–26, 78).

[87] Mate Tejfel u. a. „Improving quality of software analyser and transformer tools using specification based testing“. In: *9th Joint Conference on Mathematics and Computer Science, February 9–12, 2012, Sifok, Hungary*. 2012 (siehe S. 59).

[88] Melinda Tóth und István Bozó. „Static Analysis of Complex Software Systems Implemented in Erlang“. In: Lecture Notes in Computer Science (LNCS) 7241 (2012), S. 451–514 (siehe S. 58).

[89] Helge Toutenburg und Christian Heumann. *Deskriptive Statistik*. 6. Aufl. Springer-Verlag, 2008 (siehe S. 20, 81, 82, 86, 87).

[90] Rajesh Vasa. „Growth and Change Dynamics in Open Source Software Systems“. Diss. Melbourne, Australia, 2010 (siehe S. 81).

[91] Ernest Wallmüller. *Software Quality Engineering* (siehe S. 124).

[92] Guido Walz, Hrsg. *Lexikon der Mathematik*. Bd. 1. Spektrum Akademischer Verlag, 2001 (siehe S. 43).

[93] Guido Walz, Hrsg. *Lexikon der Mathematik*. Bd. 5. Spektrum Akademischer Verlag, 2001 (siehe S. 23).

[94] Elke Warmuth und Walter Warmuth. *Elementare Wahrscheinlichkeitsrechnung*. 1998 (siehe S. 5).

[95] Christoph Weischer. *Sozialforschung*. UVK Verlagsgesellschaft, 2007 (siehe S. 19).

[96] Robin W. Whitty, Norman E. Fenton und Agnes A. Kaposi. „Structured programming: a tutorial guide“. In: *Software & Microsystems* 3.3 (1984), S. 54–65 (siehe S. 30, 31).

[97] Horst Zuse. *A Framework of Software Measurement*. de Gruyter, 1998 (siehe S. 19, 23, 32, 33).

[98] Horst Zuse. „Resolving the Mysteries of the Halstead Measures“. vom Autor per E-Mail erhalten, 16.08.2011. 2005 (siehe S. 23).

[99] Horst Zuse. *Software Complexity : Measures and Methods*. Bd. 4. Programming complex systems. de Gruyter, 1991 (siehe S. 18–20, 22).

Anhang

A Software Measurement Ontology

Die folgenden Tabellen sind übersetzt aus García u. a. [32:636ff.], wobei in den beiden letzten Spalten jeweils der englische Begriff und eine Quelle für dessen deutsche Übersetzung angegeben ist.

[110] García u. a. [32] verwendeten für die Software Measurement Ontology die damals gültige zweite Ausgabe des *Internationales Wörterbuch der Metrologie – International Vocabulary of Basic and General Terms in Metrology* (*VIM*) [23]. Inzwischen ist die dritte Ausgabe erschienen [24]. Zur Übersetzung von Begriffen aus dem *VIM* wurde dennoch ebenfalls die zweite Ausgabe verwendet.

Tabelle A.1 – Definitionen der Begriffe in der SMO (I) (Übersetzung [vgl. 32:636])

Begriff	Oberbegriff	Definition	Quelle	engl. Begriff	Übersetzung
Informationsbedarf	Begriff	Erkenntnisse, die notwendig sind, um Vorgaben, Ziele, Risiken und Probleme zu bewältigen	ISO/IEC 15939	Information need	
Messbarer Sachverhalt	Begriff	Abstrakte Beziehung zwischen Attributen von Messobjekten und Informationsbedürfnissen	ISO/IEC 15939	Measurable concept	
Messobjekt	Begriff	Objekt, das durch Messung seiner Attribute charakterisiert werden soll	ISO/IEC 15939	Entity	[91:202]
Messobjektklasse	Begriff	Die Menge aller Messobjekte mit einer bestimmten Eigenschaft	Neu	Entity class	
Attribut	Begriff	Eine messbare physikalische oder abstrakte Eigenschaft eines Messobjekts, welche allen Messobjekten einer Messobjektklasse gemeinsam ist	nach ISO/IEC 14598	Attribute	
Qualitätsmodell	Begriff	Die Menge der messbaren Sachverhalte und ihre Beziehungen, die zusammen die Grundlage für die Festlegung von Qualitätsanforderungen und für die Bewertung der Qualität von Messobjekten einer gegebenen Messobjektklasse bilden.	nach ISO/IEC 14598		
Maß	Begriff	Festgelegter Messansatz in Verbindung mit einer Skala. (Ein Messansatz ist entweder eine Messmethode, eine Messfunktion oder ein Analysemodell.)	nach 14598 „Metrik“	Measure	
Skala	Begriff	Eine Menge von Werten mit definierten Eigenschaften.	ISO/IEC 14598	Scale	
Skalentyp	Begriff	Die Art der Beziehungen zwischen Werten der Skala.	ISO/IEC 15939	Type of scale	
Maßeinheit	Begriff	Durch Vereinbarung festgelegte spezielle Größe, mit der andere Größen gleicher Art verglichen werden, um das Verhältnis zu dieser Größe auszudrücken.	VIM	Unit of measurement	[23:21][110]
Basismaß	Maß	Ein Maß eines Attributs, welches auf keinem anderen Maß aufbaut und dessen Messansatz eine Messmethode ist.	nach ISO/IEC 14598 “direct metric”	base measure	
Abgeleitetes Maß	Maß	Ein Maß, das mittels einer Messfunktion als Messansatz von anderen Basis- oder abgeleiteten Maßen abgeleitet ist.	nach ISO/IEC 14598 „indirect metric“	Derived measure	
Indikator	Maß	Ein Maß, das mittels eines Analysemodells als Messansatz von anderen Maßen abgeleitet ist.	Neu	Indicator	

Tabelle A.2 – Definitionen der Begriffe in der SMO (II) (Übersetzung [vgl. 32:637])

Begriff	Oberbegriff	Definition	Quelle	engl. Begriff	Übersetzung nach
Messmethode	Messansatz	Allgemein beschriebene logische Abfolge von Operationen, die angewendet werden, um ein Attribut in Bezug auf eine bestimmte Skala zu quantifizieren. (Eine Messmethode ist der Messansatz eines Basismaßes.)	nach ISO/IEC 15939	Measurement method	[vgl. 24:29]
Messfunktion	Messansatz	Algorithmus oder Berechnung, durch den/die zwei oder mehr Basis- oder abgeleitete Maße verknüpft werden. (Eine Messfunktion ist der Messansatz eines abgeleiteten Maßes.)	nach ISO/IEC 15939	Measurement function	[vgl. 24:46]
Analysemodell	Messansatz	Algorithmus oder Berechnung, durch den/die zwei oder mehr Maße mit zugehörigen Entscheidungskriterien verknüpft werden. (Ein Analysemodell ist der Messansatz eines Indikators.)	nach ISO/IEC 15939	Analysis model	
Entscheidungskriterium	Begriff	Schwellwerte, Ziele oder Muster, die verwendet werden, um die Notwendigkeit von Aktionen oder weiteren Untersuchungen festzustellen, oder um die Vertrauenswürdigkeit eines Ergebnisses anzugeben.	ISO/IEC 15939	Decision criteria	
Messansatz	Begriff	Abfolge von Operationen, die darauf abzielen, den Wert eines Messergebnisses festzustellen. (Ein Messansatz ist entweder eine Messmethode, eine Messfunktion, oder ein Analysemodell.)	Neu	Measurement approach	
Messung	Begriff	Gesamtheit der Tätigkeiten, um mittels eines Messansatzes ein Messergebnis für ein bestimmtes Attribut eines Messobjekts zu ermitteln.	nach VIM	Measurement	nach [23:31]
Messergebnis	Begriff	Die Zahl oder Kategorie, die durch eine Messung einem Attribut eines Messobjekts zugeordnet wird.	nach ISO/IEC 14598 „Measure“	Measurement result	[vgl. 24:30]

B Mess- und Auswertungsumgebung

B.1 Verwendete fremde Programme und Bibliotheken

- PostgreSQL 9.1.3 als Datenbankmanagementsystem
- Bash 4.1.10 und 4.2.10, Ruby 1.8.7, GNU Awk 3.1.8, Perl 5.12.4
- TRANG: `http://www.thaiopensource.com/relaxng/trang.html`

B.2 Skripte und Programme

Die Messumgebung besteht aus einer Sammlung von Skripten und Programmen, die für diese Untersuchung entwickelt wurden. Sie sind im elektronischen Anhang zu dieser Arbeit in folgender Dateistruktur abgelegt:

- Messumgebung
  - 1 setup
    - README.txt
  - 2 database
    - createdb.sh
    - createdb.sql
  - 3a import_todo
    - (Zehn Unterverzeichnisse mit den rohen Messdaten.)
  - 4 importissues
    - ejab-issues
    - importissues.pl
  - 5 importlog
    - ejabberd_log_120317.txt
    - importlog.rb
  - 6 linecount
    - linecount.rb
    - linediff.rb
  - 7 cleanup

        - purgedb1.sql
        - purgedb2.sql
        - purgedb3.sql
        - unify_names.sql
      - 8 readpatch
        - ejabberd_revisions_until_2011-12-30.txt
        - fixlinks.sql
        - issuecodelink.rb
      - 9 indirect measures
        - create_zeros.sql
        - fill_measures.sql
        - get_indirect_measures1.sql
        - get_indirect_measures2.sql
        - get_indirect_measures3.sql
      - 10 export
        - checkmeasures.r
        - describemeasures.r
        - exportmeasures.rb
        - measures.r
  - importmeasures.rb
  - utils.rb

B.2.1 Skripte zur Überprüfung der Werkzeugvalidität

B.2.1.1 Anzahl der Module

Skript B.1 – Bash-Skript zur Ausgabe der Anzahl von Erlang-Dateien pro ejabberd-Release

```
1 #!/usr/bin/env bash
-
- # get number of Erlang source files for all subdirectories (
    non-recursively)
-
5 for d in `find . -maxdepth 1 -type d | sort`
- do
-   echo "$d: `find $d -name "*.erl" -print | wc`"
- done
```

Skript B.2 – SQL-Skript zur Ausgabe der Anzahl geladener Module pro ejabberd-Release

```
1 select m.revision, count(distinct m.mod)
- from dmeasure m, m_loaded l
- where m.revision like 'r%'
```

```
        and l.revision = m.revision
        and l.mod = m.mod
        and l.loaded = true
group by m.revision
order by m.revision
```

B.2.1.2 Anzahl der Funktionen

Skript B.3 – SQL-Abfrage zur Ausgabe der Anzahl von Funktionen, die in die Messumgebung importiert wurden

```
select m.revision, count(distinct (m.mod,m.fun,m.arity))
from dmeasure m
where m.revision like 'r%'
      and (m.revision, m.mod, m.fun, m.arity) not in
          (select nd.revision, nd.mod, nd.fun, nd.arity
           from mf_notdefined nd
           where nd.revision = m.revision)
group by m.revision
order by m.revision
```

Skript B.4 – SQL-Abfrage zur Ausgabe der Messergebnisse für die Anzahl der Funktionen pro EJABBERD-Release

```
select revision, sum(value)
from dmeasure
where revision like 'r%' and measure = 'number_of_fun'
group by revision
order by revision
```

B.2.1.3 Anzahl der Zeilen

Skript B.5 – RUBY-Skript zum Zählen von Zeilen. Ausschnitt – vollständige Version im digitalen Anhang, `code/6 linecount/linecount.rb`

```
while (line = file.gets)
  if line.match(/^$/) then blanks += 1 end
  if line.match(/^[ \t]+$/) then quasiblanks += 1 end
  if line.match(/\S+/) then loc += 1 end
  if line.match(/^\s*%/) then cloc += 1 end
  if line.match(/\w+%/) then pcloc += 1 end
end
```

B.2.2 Skripte zur Verknüpfung von Fällen und Code

Skript B.6 – RUBY-Skript zur Analyse von PATCH-Dateien. Ausschnitt – vollständige Version im digitalen Anhang, `code/8 readpatch/issuecodelink.rb`

```
# <id> is just used as source identifier
```

```
def getPatchTarget(ikey, revID, id, filename, note)
  @log.info("read patch for #{revID}: #{filename}...\n")

  patchfile = File.new(filename, "r")
  while (line = patchfile.gets)
    if m = line.match(/^\-\-\- (\S+)/)
      mod = File.basename(m[1], ".erl")
    elsif m = line.match(/^@@ \-(\d+),?(\d*).*\+(\d+),?(\d*)
       .* @@/)
      startindex, endindex = 1, 2
      startline = m[startindex].to_i()
      endline = m[startindex].to_i() + m[endindex].to_i() - 1

      targetFuns = getFunctions(revID, mod, startline,
         endline)
      print "FUNS: #{targetFuns.to_s()}\n"
      targetFuns.each { |r, m, f, a|
        addIssueCodeLink(ikey, revID, m, f, a, "#{note},#{id}
           /#{filename}")
      }
    end
  end
  patchfile.close
end
```

C Ergänzende Materialien

C.1 Nicht überprüfte Valdierungskriterien

„Actionability“ und Konstruktivität, #2,#11 Ein Maß soll als Entscheidungsgrundlage für Projektmanager taugen („*Actionability*“) [65:15] beziehungsweise Forschern helfen, Softwarequalität zu verstehen (*Konstruktivität*) [65:17]. „Actionability“ vermischt die Eigenschaft des Maßes, eine relevante Qualitätseigenschaft zutreffend abzubilden mit der generellen Fähigkeit des Managers, rational zu entscheiden. Letztere kann im Rahmen dieser Arbeit nicht untersucht werden. ·Konstruktivität scheint zu abstrakt, um überprüfbar zu sein. Es wird unter anderem durch ·Assoziation und ·Trennschärfe konkretisiert.

Angemessene Granularität, #4 Ein Maß soll nicht zu fein und nicht zu grob abgestuft sein [65:15]. Dies wird hier nicht betrachtet, weil es vermutlich stark von den Bedürfnissen einzelner Anwender abhängt, was „zu fein“ und was „zu grob“ ist.

Attributvalidität, #6 Messwerte müssen die zu messende Eigenschaft korrekt wiedergeben [65:16]. Dies ist nur eine andere Formulierung der Definition von ·interner Validität.

Kausalmodellvalidität und Kausalbeziehungsvalidität, #7,#8 Die Kriterien beziehen sich darauf, ob ein Maß als Teil eines Vorhersagemodells verwendet werden kann (*Kausalmodellvalidität*), beziehungsweise ob darüber hinaus eine Kausalbeziehung zu einer externen Qualitätseigenschaft nachgewiesen werden kann (*Kausalbeziehungsvalidität*) [65:16]. Die Untersuchung von Vorhersagemodellen würde den Rahmen dieser Arbeit übersteigen. Zudem ist die Unterscheidung der beiden Kriterien unklar.

Inhaltliche Validität, #9 Ein Maß soll den interessierenden Qualitätsbereich vollständig abdecken [65:16].

Ökonomische Produktivität und Benutzbarkeit, #15/#47 Der Messaufwand soll nicht den Nutzen übersteigen (*Ökonomische Produktivität*) [65:17]. Dieses Kriterium wird von Meneely, Smith und Williams [65:24] als Form interner, theoretischer Validität eingeordnet. Die Kosten der Messung können jedoch nur empirisch festgestellt werden und hängen zudem von der Effizienz der Implementierung ab, der Nutzen ist ein externes Phänomen – daher erscheint eine Einordnung als externe, empirische Validität beziehungsweise Konstruktvalidität sinnvoller. Benutzbarkeit (#47) ist in der Definition von Meneely, Smith und Williams [65:23] synonym zu ökonomischer Produktivität.

Monotones Wachstum, #21 Der gemeinsame Messwert zweier Komponenten darf nie kleiner sein als die einzelnen Messwerte der Komponenten [65:18]. Durch Verallgemeinerung ***einer*** intuitiven Vorstellung von Softwaremaßen auf möglicherweise ganz andersartige Eigenschaften schränkt dieses Kriterium die Homomorphieeigenschaft aller Maße unnötig ein.

Interaktionssensitivität, #22 Unterschiedliche Arten der Interaktion (oder Verknüpfung) zweier Komponenten sollen zu unterschiedlichen Messwerten führen [65:18]. Dieses Kriterium kann, weil intern und theoretisch, separat untersucht werden und wird daher hier ausgespart.

Interne Konsistenz, #23 Die Komponenten eines ·abgeleiteten Maßes sollen die selbe Eigenschaft erfassen und zusammenhängen [65:19]. Diese Eigenschaft ist das Gegenteil von ·Faktorenunabhängigkeit und eine unnötige Verschärfung von ·Dimensionskonsistenz. Demnach wäre der Quotient aus Weg und Zeit kein valides (Geschwindigkeits-)Maß, da zwei unterschiedliche Eigenschaften verknüpft werden.

Monotonität, #25 Die Verknüpfung zweier Komponenten darf keinen kleineren Messwert ergeben als die einzelnen Komponenten [65:19]. Siehe ·Monotones Wachstum.

Zuverlässigkeit, #26 Die Messung soll präzise und reproduzierbar sein [65:19]. Dies ist eine Zusammenfassung von ·Interne Konsistenz und ·Stabilität.

Nicht-Kolinearität, #27 Die Korrelation eines internen Maßes mit einem externen Maß soll erhalten bleiben, wenn andere Einflussgrößen kontrolliert werden [65:19]. Dies ist ein Spezialfall von ·Verbesserungsvalidität.

Missbrauchssicherheit, #28 Messwerte dürfen nicht gezielt durch irrelevante Änderungen am Messobjekt manipuliert werden können [65:19]. Diese Eigenschaft folgt aus ·interner Validität, bietet aber keinen konkreten Ansatz zu deren Nachweis.

Nicht-Gleichförmigkeit, #29 Ein Maß soll für zwei unterschiedliche Messobjekte auch verschiedene Messwerte ergeben [65:19]. Insofern die Messobjekte unterschiedlich ***bezüglich der gemessenen Eigenschaft*** sein sollen, ist dieses Kriterium nur eine andere Formulierung der Homomorphieeigenschaft eines ·Maßes. Abseits dessen ist es durchaus möglich, dass unterschiedliche Messobjekte in einzelnen Aspekten gleich sind, mit entsprechend gleichen Messwerten.

Permutationsvalidität, #31 Messwerte sollen durch die Veränderung der Reihenfolge der Programmanweisungen beeinflusst werden [65:31]. Dieses Kriterium wurde für die Pseudoeigenschaft „Komplexität“ vorgeschlagen (siehe Abschnitt 2.1.1, S. 18). Eine allgemeine Anwendbarkeit ist nicht ersichtlich.

Eignung für Vorhersagen und als Teil eines Vorhersagesystems, #32,#33 Eine externe Qualitätseigenschaft soll von dem Maß selbst (#32) beziehungsweise als Teil eines Vorhersagemodells (#33) mit ausreichender Genauigkeit vorhergesagt werden können [65:20]. In dieser Arbeit werden Vorhersagemodelle nicht untersucht.

Prozess- oder Produktrelevanz, #34 Ein Maß soll durch Anpassung auch in einem neuen Anwendungsbereich valide bleiben [65:20]. Diese Eigenschaft wird durch die erwähnte kontinuierliche Validierung überprüft.

Rangkonsistenz, #36 Ein Maß soll die gleiche Rangordnung ergeben wie eine externe Qualitätseigenschaft (beziehungsweise deren Maß) [65:20]. Dies ist ein Spezialfall von ·Assoziation beziehungsweise ·Paralleler Veränderung.

Transformationsinvarianz und Robustheit gegen Umbenennung, #44,#37 Messwerte sollen durch Transformation (#44) des Messobjektes, insbesondere Umbenennung von Bezeichnern (#37), nicht beeinflusst werden [65:20,22]. Dies ist eine spezielle Sicht von ·interner Validität, die auf der unbewiesenen Annahme beruht, dass die Namen von Bezeichnern (#37) oder, noch gewagter, die Struktur der Software (#44) für die Softwarequalität irrelevant sind. ·Robustheit gegen Umbenennung ist ein Spezialfall von ·Transformationsinvarianz.

Repräsentationsbedingung, #39 Als *Repräsenationsbedingung* wird die Homomorphieeigenschaft eines ·Maßes bezeichnet [65:21]. Meneely, Smith und Williams [65:21] behaupten, dies bedeute, dass ***jegliche*** Eigenschaft des formalen Relationensystems eine Entsprechung im empirischen Relationensystem haben müsse. Bei der Messung geht es jedoch darum, empirische Relationen durch entsprechende formale Relationen wiederzugeben. Je nachdem, ***welche*** empirischen Relationen in den Messwerten erhalten bleiben, besitzt das Maß einen anderen ·Skalentyp (siehe **??**, S. ??).

Einheitenvalidität, #46 Die verwendete Einheit soll der zu messenden Eigenschaft angemessen sein [65:22]. Insofern es sich darum handelt, dass die Messwerte in dieser Einheit die empirischen Relationen wiedergeben, ist dies nur eine andere Formulierung für ·interne Validität. Siehe auch unter ·Granularität.

D Ergänzende Tabellen

D.1 Zu Abschnitt 4.3.3, S. 81

D.2 Zu Abschnitt 4.3.6, S. 87

Tabelle D.1 – Zentrale Momente der internen Maße für Funktionen. $\mathfrak{z}_2$ ist die Streuung, $\mathfrak{z}_3$ die Schiefe und $\mathfrak{z}_4$ die Wölbung [64:65].

Maß	$\mathfrak{z}_0$	$\mathfrak{z}_1$	$\mathfrak{z}_2$	$\mathfrak{z}_3$	$\mathfrak{z}_4$
LOC_F	1.0	0.0	1.0	7.3	84.5
MaxCall_F	1.0	-0.0	1.0	2.0	7.1
MaxCase_F	1.0	-0.0	1.0	1.9	7.5
MaxNest_F	1.0	0.0	1.0	1.6	5.8
$\text{NumClauses}_\text{F}$	1.0	-0.0	1.0	9.8	124.6
$\text{RecBranch}_\text{F}$	1.0	-0.0	1.0	32.6	1435.7
FanIn_F	1.0	0.0	1.0	25.9	957.3
FanOut_F	1.0	0.0	1.0	3.8	30.1
NumAnon_F	1.0	-0.0	1.0	5.7	59.1
NumSend_F	1.0	-0.0	1.0	20.8	583.6
Returns_F	1.0	0.0	1.0	9.8	160.1
Cyc_F	1.0	0.0	1.0	8.8	120.7

Tabelle D.2 – Zentrale Momente der internen Maße für Module. $\mathfrak{z}_2$ ist die Streuung, $\mathfrak{z}_3$ die Schiefe und $\mathfrak{z}_4$ die Wölbung [64:65].

Maß	$\mathfrak{z}_0$	$\mathfrak{z}_1$	$\mathfrak{z}_2$	$\mathfrak{z}_3$	$\mathfrak{z}_4$
NCLOC_M	1.0	-0.0	1.0	3.6	19.1
CLOC_M	1.0	0.0	1.0	4.3	27.7
LOC_M	1.0	0.0	1.0	3.5	18.6
LOCF_M	1.0	-0.0	1.0	3.6	18.7
Included_M	1.0	-0.0	1.0	0.3	2.6
Imported_M	1.0	0.0	1.0	7.6	59.2
NumMac_M	1.0	-0.0	1.0	3.0	12.9
NumRec_M	1.0	0.0	1.0	2.4	12.0
NumFun_M	1.0	0.0	1.0	2.9	15.4
CallsIn_M	1.0	-0.0	1.0	8.4	89.6
CallsOut_M	1.0	0.0	1.0	2.8	12.9
Calls_M	1.0	-0.0	1.0	3.8	22.8
$\text{NumClauses}_\text{M}$	1.0	0.0	1.0	2.8	12.4
Returns_M	1.0	-0.0	1.0	3.3	17.1
NumAnon_M	1.0	-0.0	1.0	5.9	49.3
NumSend_M	1.0	0.0	1.0	5.9	49.5
$\text{RecBranch}_\text{M}$	1.0	-0.0	1.0	5.3	40.2
MaxCall_M	1.0	-0.0	1.0	0.9	3.2
MaxCase_M	1.0	-0.0	1.0	0.4	2.6
MaxNest_M	1.0	-0.0	1.0	0.3	2.7
$\text{NumNonRec}_\text{M}$	1.0	-0.0	1.0	2.8	14.8
$\text{NumTrivRec}_\text{M}$	1.0	0.0	1.0	3.3	16.6
$\text{NumNonTrivRec}_\text{M}$	1.0	0.0	1.0	0.6	2.3
WMC(CYC)_M	1.0	-0.0	1.0	3.7	21.1
RFC_M	1.0	-0.0	1.0	2.9	14.2
AvgLOC_M	1.0	0.0	1.0	2.5	12.9
AvgCyc_M	1.0	-0.0	1.0	2.2	12.1
AvgCalls_M	1.0	0.0	1.0	3.3	20.2
AvgOut_M	1.0	0.0	1.0	-1.4	3.6
CBO_M	1.0	0.0	1.0	3.7	27.4
InstF_M	1.0	0.0	1.0	-1.0	2.9
InstM_M	1.0	0.0	1.0	-0.6	2.7

Tabelle D.3 – Einige Kenngrößen der internen Maße für Funktionen. $x_{\min}$ und $x_{\max}$ sind der kleinste beziehungsweise größte Messwert, $\tilde{d}_{0,5}$ ist die mittlere absolute Abweichung vom Median. $x_{0,25}$, $x_{0,5}$ und $x_{0,75}$ sind unteres Quartil, Median und oberes Quartil. Werte bis x_{lo} und ab x_{hi} gelten als „Ausreißer“ (siehe Abschnitt 4.3.2.1, S. 81.

Maß	$x_{\min}$	$x_{\max}$	$\tilde{d}_{0.5}$	$\tilde{x}_{\text{lo}}$	$\tilde{x}_{0.25}$	$\tilde{x}_{0.5}$	$\tilde{x}_{0.75}$	$\tilde{x}_{\text{hi}}$
LOC_{F}	1.0	504.0	11.9	-17.0	4.0	9.0	18.0	39.0
$\text{MaxCall}_{\text{F}}$	0.0	30.0	3.2	-6.5	1.0	2.0	6.0	13.5
$\text{MaxCase}_{\text{F}}$	0.0	6.0	0.6	-1.5	0.0	0.0	1.0	2.5
$\text{MaxNest}_{\text{F}}$	0.0	8.0	0.9	-1.5	0.0	0.0	1.0	2.5
$\text{NumClauses}_{\text{F}}$	1.0	41.0	0.6	1.0	1.0	1.0	1.0	1.0
$\text{RecBranch}_{\text{F}}$	0.0	87.0	0.2	0.0	0.0	0.0	0.0	0.0
FanIn_{F}	0.0	761.0	2.6	-3.0	0.0	1.0	2.0	5.0
FanOut_{F}	0.0	58.0	2.3	-3.5	1.0	2.0	4.0	8.5
$\text{NumAnon}_{\text{F}}$	0.0	15.0	0.3	0.0	0.0	0.0	0.0	0.0
$\text{NumSend}_{\text{F}}$	0.0	10.0	0.0	0.0	0.0	0.0	0.0	0.0
$\text{Returns}_{\text{F}}$	1.0	112.0	1.5	-2.0	1.0	1.0	3.0	6.0
Cyc_{F}	1.0	112.0	2.0	-2.0	1.0	2.0	3.0	6.0

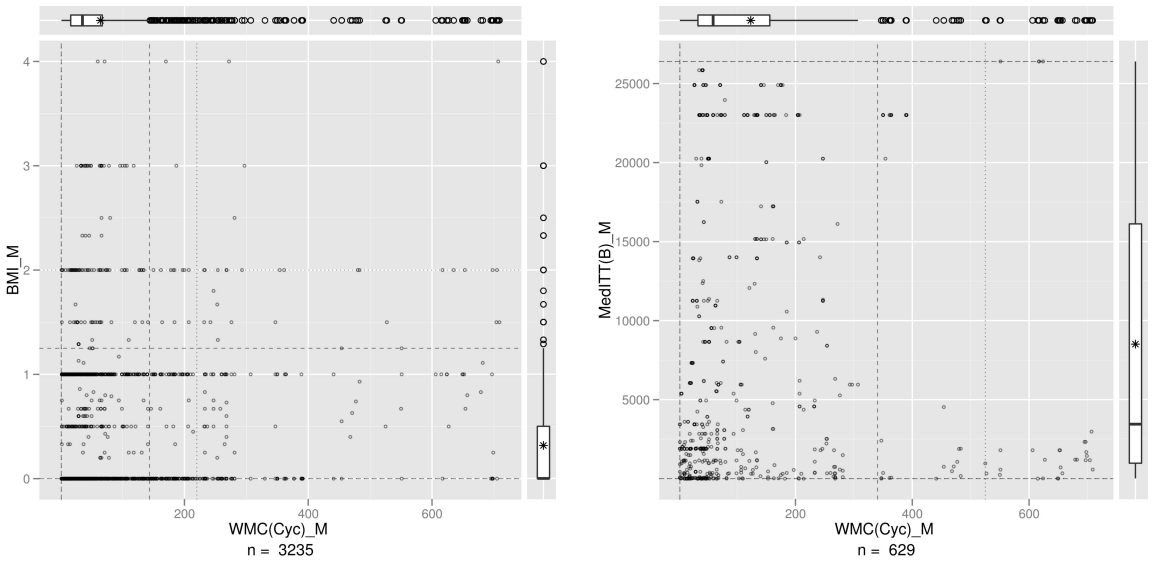


Abbildung D.1 – Streudiagramme für $\text{WMC}(\text{Cyc})_{\text{M}}$ und BMI_{M} (links) beziehungsweise $\text{MedITT}(\text{B})_{\text{M}}$ (rechts)

Tabelle D.4 – Einige Kenngrößen der internen Maße für Module. $x_{\min}$ und $x_{\max}$ sind der kleinste beziehungsweise größte Messwert, $\tilde{d}_{0,5}$ ist die mittlere absolute Abweichung vom Median. $x_{0,25}$, $x_{0,5}$ und $x_{0,75}$ sind unteres Quartil, Median und oberes Quartil. Werte bis x_{lo} und ab x_{hi} gelten als „Ausreißer“ (siehe Abschnitt 4.3.2.1, S. 81.

Maß	$x_{\min}$	$x_{\max}$	$\tilde{d}_{0.5}$	$\tilde{x}_{\text{lo}}$	$\tilde{x}_{0.25}$	$\tilde{x}_{0.5}$	$\tilde{x}_{0.75}$	$\tilde{x}_{\text{hi}}$
NCLOC_{M}	4.0	3699.0	238.6	-349.5	57.0	148.0	328.0	734.5
CLOC_{M}	0.0	543.0	32.1	-41.5	20.0	32.0	61.0	122.5
LOC_{M}	15.0	3862.0	273.2	-352.5	120.0	202.0	435.0	907.5
LOCF_{M}	4.0	3724.0	256.7	-367.5	84.0	163.0	385.0	836.5
$\text{INCLUDED}_{\text{M}}$	0.0	5.0	1.0	-0.5	1.0	2.0	2.0	3.5
$\text{IMPORTED}_{\text{M}}$	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
NUMMAC_{M}	0.0	26.0	2.4	-4.5	0.0	0.0	3.0	7.5
NUMREC_{M}	0.0	8.0	0.7	-1.5	0.0	0.0	1.0	2.5
NUMFUN_{M}	1.0	163.0	13.5	-21.0	9.0	16.0	29.0	59.0
$\text{CALLSIN}_{\text{M}}$	0.0	934.0	18.6	-12.0	0.0	2.0	8.0	20.0
$\text{CALLSOUT}_{\text{M}}$	0.0	402.0	34.5	-63.0	12.0	30.0	62.0	137.0
CALLS_{M}	0.0	1070.0	64.3	-101.5	23.0	45.0	106.0	230.5
$\text{NUMCLAUSES}_{\text{M}}$	1.0	271.0	23.3	-30.5	10.0	21.0	37.0	77.5
$\text{RETURNS}_{\text{M}}$	1.0	486.0	37.0	-46.0	14.0	32.0	54.0	114.0
$\text{NUMANON}_{\text{M}}$	0.0	126.0	5.4	-10.5	0.0	2.0	7.0	17.5
$\text{NUMSEND}_{\text{M}}$	0.0	20.0	0.7	0.0	0.0	0.0	0.0	0.0
$\text{RECBRANCH}_{\text{M}}$	0.0	101.0	4.0	-6.0	0.0	1.0	4.0	10.0
$\text{MAXCALL}_{\text{M}}$	0.0	30.0	5.1	-8.0	4.0	7.0	12.0	24.0
$\text{MAXCASE}_{\text{M}}$	0.0	6.0	1.2	-2.0	1.0	2.0	3.0	6.0
$\text{MAXNEST}_{\text{M}}$	0.0	8.0	1.3	-1.0	2.0	3.0	4.0	7.0
$\text{NUMNONREC}_{\text{M}}$	1.0	143.0	12.3	-21.5	7.0	14.0	26.0	54.5
$\text{NUMTRIVREC}_{\text{M}}$	1.0	20.0	1.8	-3.5	1.0	2.0	4.0	8.5
$\text{NUMNONTRIVREC}_{\text{M}}$	1.0	5.0	0.9	0.5	2.0	2.0	3.0	4.5
$\text{WMC(CYC)}_{\text{M}}$	1.0	709.0	47.6	-60.5	16.0	35.0	67.0	143.5
RFC_{M}	1.0	548.0	46.4	-77.0	22.0	45.0	88.0	187.0
AVGLOC_{M}	0.1	86.5	7.0	-7.2	8.2	12.1	18.5	33.9
AVGCYC_{M}	0.0	13.0	1.0	-0.7	1.7	2.2	3.2	5.6
$\text{AVGCALLS}_{\text{M}}$	0.0	25.7	1.6	-1.6	1.6	2.6	3.7	6.8
AVGOUT_{M}	0.0	1.0	0.2	0.2	0.7	0.9	1.0	1.5
CBO_{M}	0.0	174.0	9.5	-24.0	0.0	8.0	16.0	40.0
INSTF_{M}	0.0	1.0	0.2	-0.1	0.6	0.9	1.0	1.6
INSTM_{M}	0.0	1.0	0.2	-0.1	0.5	0.7	0.9	1.5

Tabelle D.5 – Zentrale Momente der externen Maße für Funktionen. $\mathfrak{z}_2$ ist die Streuung, $\mathfrak{z}_3$ die Schiefe und $\mathfrak{z}_4$ die Wölbung [64:65].

Maß	$\mathfrak{z}_0$	$\mathfrak{z}_1$	$\mathfrak{z}_2$	$\mathfrak{z}_3$	$\mathfrak{z}_4$
NumIss(B)$_F$	1.0	-0.0	1.0	6.6	62.6
NumIss(N)$_F$	1.0	0.0	1.0	4.5	26.2
NumIss(T)$_F$	1.0	-0.0	1.0	14.8	228.4
NumIss(I)$_F$	1.0	0.0	1.0	3.8	24.0
StartIss(B)$_F$	1.0	-0.0	1.0	9.0	116.7
StartIss(N)$_F$	1.0	0.0	1.0	8.8	88.5
StartIss(T)$_F$	1.0	0.0	1.0	19.1	365.9
StartIss(I)$_F$	1.0	-0.0	1.0	7.5	72.8
EndIss(B)$_F$	1.0	-0.0	1.0	10.7	154.5
EndIss(N)$_F$	1.0	0.0	1.0	10.2	118.8
EndIss(T)$_F$	1.0	0.0	1.0	25.7	661.6
EndIss(I)$_F$	1.0	0.0	1.0	7.8	79.5
MedITT$_F$	1.0	0.0	1.0	0.7	2.1
MedITT(B)$_F$	1.0	-0.0	1.0	0.8	1.9
MedITT(N)$_F$	1.0	0.0	1.0	0.3	2.1
MedITT(T)$_F$	1.0	-0.0	1.0	1.6	3.8
MedITT(I)$_F$	1.0	0.0	1.0	0.7	1.8
StartRate(B)$_F$	1.0	-0.0	1.0	10.5	139.8
StartRate(N)$_F$	1.0	-0.0	1.0	16.1	368.4
StartRate(T)$_F$	1.0	-0.0	1.0	36.7	1659.6
StartRate(I)$_F$	1.0	0.0	1.0	12.2	201.8
EndRate(B)$_F$	1.0	0.0	1.0	17.5	505.4
EndRate(N)$_F$	1.0	0.0	1.0	19.1	459.7
EndRate(T)$_F$	1.0	-0.0	1.0	45.2	2267.2
EndRate(I)$_F$	1.0	0.0	1.0	9.0	105.2
ImproveRate$_F$	1.0	-0.0	1.0	5.0	26.4
BMI$_F$	1.0	0.0	1.0	5.3	37.2
BugRate$_F$	1.0	0.0	1.0	4.6	23.1

Tabelle D.6 – Zentrale Momente der externen Maße für Module. $\mathfrak{z}_2$ ist die Streuung, $\mathfrak{z}_3$ die Schiefe und $\mathfrak{z}_4$ die Wölbung [64:65].

Maß	$\mathfrak{z}_0$	$\mathfrak{z}_1$	$\mathfrak{z}_2$	$\mathfrak{z}_3$	$\mathfrak{z}_4$
NumIss(B)$_M$	1.0	0.0	1.0	6.6	78.7
NumIss(N)$_M$	1.0	0.0	1.0	3.1	15.0
NumIss(T)$_M$	1.0	0.0	1.0	2.2	7.5
NumIss(I)$_M$	1.0	0.0	1.0	4.0	27.2
StartIss(B)$_M$	1.0	-0.0	1.0	10.2	183.0
StartIss(T)$_M$	1.0	-0.0	1.0	2.1	6.6
StartIss(I)$_M$	1.0	0.0	1.0	5.9	57.5
EndIss(B)$_M$	1.0	0.0	1.0	9.4	163.5
EndIss(N)$_M$	1.0	0.0	1.0	5.8	54.5
EndIss(T)$_M$	1.0	-0.0	1.0	1.9	4.9
EndIss(I)$_M$	1.0	0.0	1.0	4.8	30.7
MedITT$_M$	1.0	0.0	1.0	0.9	2.9
MedITT(B)$_M$	1.0	0.0	1.0	0.8	2.1
MedITT(N)$_M$	1.0	-0.0	1.0	0.3	2.0
MedITT(T)$_M$	1.0	0.0	1.0	2.0	5.8
MedITT(I)$_M$	1.0	-0.0	1.0	0.6	2.1
StartRate(B)$_M$	1.0	-0.0	1.0	6.8	75.9
StartRate(N)$_M$	1.0	-0.0	1.0	6.1	51.8
StartRate(T)$_M$	1.0	0.0	1.0	3.3	13.3
StartRate(I)$_M$	1.0	0.0	1.0	5.3	38.6
EndRate(B)$_M$	1.0	-0.0	1.0	7.1	79.2
EndRate(N)$_M$	1.0	-0.0	1.0	6.9	64.9
EndRate(T)$_M$	1.0	-0.0	1.0	3.5	14.3
EndRate(I)$_M$	1.0	-0.0	1.0	5.2	36.9
ImproveRate$_M$	1.0	-0.0	1.0	2.6	8.3
BMI$_M$	1.0	-0.0	1.0	2.1	8.4
BugRate$_M$	1.0	0.0	1.0	2.4	7.8

Tabelle D.7 – Einige Kenngrößen der externen Maße für Funktionen. $x_{\min}$ und $x_{\max}$ sind der kleinste beziehungsweise größte Messwert, $\tilde{d}_{0,5}$ ist die mittlere absolute Abweichung vom Median. $x_{0,25}$, $x_{0,5}$ und $x_{0,75}$ sind unteres Quartil, Median und oberes Quartil. Werte bis x_{lo} und ab x_{hi} gelten als „Ausreißer“ (siehe Abschnitt 4.3.2.1, S. 81.

Maß	$x_{\min}$	$x_{\max}$	$\tilde{d}_{0.5}$	$\tilde{x}_{\text{lo}}$	$\tilde{x}_{0.25}$	$\tilde{x}_{0.5}$	$\tilde{x}_{0.75}$	$\tilde{x}_{\text{hi}}$
NumIss(B)$_{\text{F}}$	0.0	7.0	0.1	0.0	0.0	0.0	0.0	0.0
NumIss(N)$_{\text{F}}$	1.0	4.0	0.1	1.0	1.0	1.0	1.0	1.0
NumIss(T)$_{\text{F}}$	0.0	2.0	0.0	0.0	0.0	0.0	0.0	0.0
NumIss(I)$_{\text{F}}$	0.0	7.0	0.1	0.0	0.0	0.0	0.0	0.0
StartIss(B)$_{\text{F}}$	0.0	5.0	0.0	0.0	0.0	0.0	0.0	0.0
StartIss(N)$_{\text{F}}$	0.0	3.0	0.0	0.0	0.0	0.0	0.0	0.0
StartIss(T)$_{\text{F}}$	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
StartIss(I)$_{\text{F}}$	0.0	4.0	0.0	0.0	0.0	0.0	0.0	0.0
EndIss(B)$_{\text{F}}$	0.0	6.0	0.0	0.0	0.0	0.0	0.0	0.0
EndIss(N)$_{\text{F}}$	0.0	2.0	0.0	0.0	0.0	0.0	0.0	0.0
EndIss(T)$_{\text{F}}$	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
EndIss(I)$_{\text{F}}$	0.0	4.0	0.0	0.0	0.0	0.0	0.0	0.0
MedITT$_{\text{F}}$	0.1	38591.9	9592.3	-24190.0	4128.4	9830.8	23007.3	51325.7
MedITT(B)$_{\text{F}}$	0.1	26396.7	7255.9	-19831.1	1166.1	3921.1	15164.2	36161.3
MedITT(N)$_{\text{F}}$	0.2	34340.6	8399.9	-19561.6	3683.2	14613.0	19179.8	42424.6
MedITT(T)$_{\text{F}}$	0.2	13450.6	2574.7	-2336.3	47.9	47.9	1637.3	4021.5
MedITT(I)$_{\text{F}}$	0.1	38591.9	10120.1	-28692.3	5022.8	9830.8	27499.5	61214.6
StartRate(B)$_{\text{F}}$	0.0	1.5	0.0	0.0	0.0	0.0	0.0	0.0
StartRate(N)$_{\text{F}}$	0.0	1.5	0.0	0.0	0.0	0.0	0.0	0.0
StartRate(T)$_{\text{F}}$	0.0	1.5	0.0	0.0	0.0	0.0	0.0	0.0
StartRate(I)$_{\text{F}}$	0.0	1.5	0.0	0.0	0.0	0.0	0.0	0.0
EndRate(B)$_{\text{F}}$	0.0	2.9	0.0	0.0	0.0	0.0	0.0	0.0
EndRate(N)$_{\text{F}}$	0.0	1.5	0.0	0.0	0.0	0.0	0.0	0.0
EndRate(T)$_{\text{F}}$	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
EndRate(I)$_{\text{F}}$	0.0	1.5	0.0	0.0	0.0	0.0	0.0	0.0
ImproveRate$_{\text{F}}$	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
BMI$_{\text{F}}$	0.0	4.0	0.1	0.0	0.0	0.0	0.0	0.0
BugRate$_{\text{F}}$	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0

Tabelle D.8 – Einige Kenngrößen der externen Maße für Module. $x_{\min}$ und $x_{\max}$ sind der kleinste beziehungsweise größte Messwert, $\tilde{d}_{0,5}$ ist die mittlere absolute Abweichung vom Median. $x_{0,25}$, $x_{0,5}$ und $x_{0,75}$ sind unteres Quartil, Median und oberes Quartil. Werte bis x_{lo} und ab x_{hi} gelten als „Ausreißer“ (siehe Abschnitt 4.3.2.1, S. 81.

Maß	$x_{\min}$	$x_{\max}$	$\tilde{d}_{0.5}$	$\tilde{x}_{\text{lo}}$	$\tilde{x}_{0.25}$	$\tilde{x}_{0.5}$	$\tilde{x}_{0.75}$	$\tilde{x}_{\text{hi}}$
NumIss(B)$_{\text{M}}$	0.0	22.0	0.4	0.0	0.0	0.0	0.0	0.0
NumIss(N)$_{\text{M}}$	1.0	9.0	0.5	-0.5	1.0	1.0	2.0	3.5
NumIss(T)$_{\text{M}}$	0.0	3.0	0.2	0.0	0.0	0.0	0.0	0.0
NumIss(I)$_{\text{M}}$	0.0	14.0	0.5	-1.5	0.0	0.0	1.0	2.5
StartIss(B)$_{\text{M}}$	0.0	18.0	0.2	0.0	0.0	0.0	0.0	0.0
StartIss(T)$_{\text{M}}$	0.0	2.0	0.2	0.0	0.0	0.0	0.0	0.0
StartIss(I)$_{\text{M}}$	0.0	9.0	0.1	0.0	0.0	0.0	0.0	0.0
EndIss(B)$_{\text{M}}$	0.0	18.0	0.2	0.0	0.0	0.0	0.0	0.0
EndIss(N)$_{\text{M}}$	0.0	6.0	0.1	0.0	0.0	0.0	0.0	0.0
EndIss(T)$_{\text{M}}$	0.0	2.0	0.2	0.0	0.0	0.0	0.0	0.0
EndIss(I)$_{\text{M}}$	0.0	5.0	0.1	0.0	0.0	0.0	0.0	0.0
MedITT$_{\text{M}}$	0.0	38591.9	8117.4	-22901.5	648.9	6417.8	16349.1	39899.5
MedITT(B)$_{\text{M}}$	0.0	26396.7	7110.0	-21099.4	645.5	3265.8	15142.1	36887.0
MedITT(N)$_{\text{M}}$	0.2	34340.6	7882.3	-19561.6	3683.2	12853.8	19179.8	42424.6
MedITT(T)$_{\text{M}}$	0.1	13450.6	1490.2	-208.5	9.2	72.7	154.4	372.2
MedITT(I)$_{\text{M}}$	0.1	38591.9	9394.9	-25833.9	5022.8	10028.9	25594.0	56450.7
StartRate(B)$_{\text{M}}$	0.0	3.6	0.1	0.0	0.0	0.0	0.0	0.0
StartRate(N)$_{\text{M}}$	0.0	1.5	0.0	0.0	0.0	0.0	0.0	0.0
StartRate(T)$_{\text{M}}$	0.0	1.5	0.1	0.0	0.0	0.0	0.0	0.0
StartRate(I)$_{\text{M}}$	0.0	1.5	0.0	0.0	0.0	0.0	0.0	0.0
EndRate(B)$_{\text{M}}$	0.0	3.9	0.1	0.0	0.0	0.0	0.0	0.0
EndRate(N)$_{\text{M}}$	0.0	1.9	0.0	0.0	0.0	0.0	0.0	0.0
EndRate(T)$_{\text{M}}$	0.0	1.0	0.1	0.0	0.0	0.0	0.0	0.0
EndRate(I)$_{\text{M}}$	0.0	1.6	0.0	0.0	0.0	0.0	0.0	0.0
ImproveRate$_{\text{M}}$	0.0	1.0	0.1	0.0	0.0	0.0	0.0	0.0
BMI$_{\text{M}}$	0.0	4.0	0.3	-1.2	0.0	0.0	0.8	2.1
BugRate$_{\text{M}}$	0.0	1.0	0.1	0.0	0.0	0.0	0.0	0.0

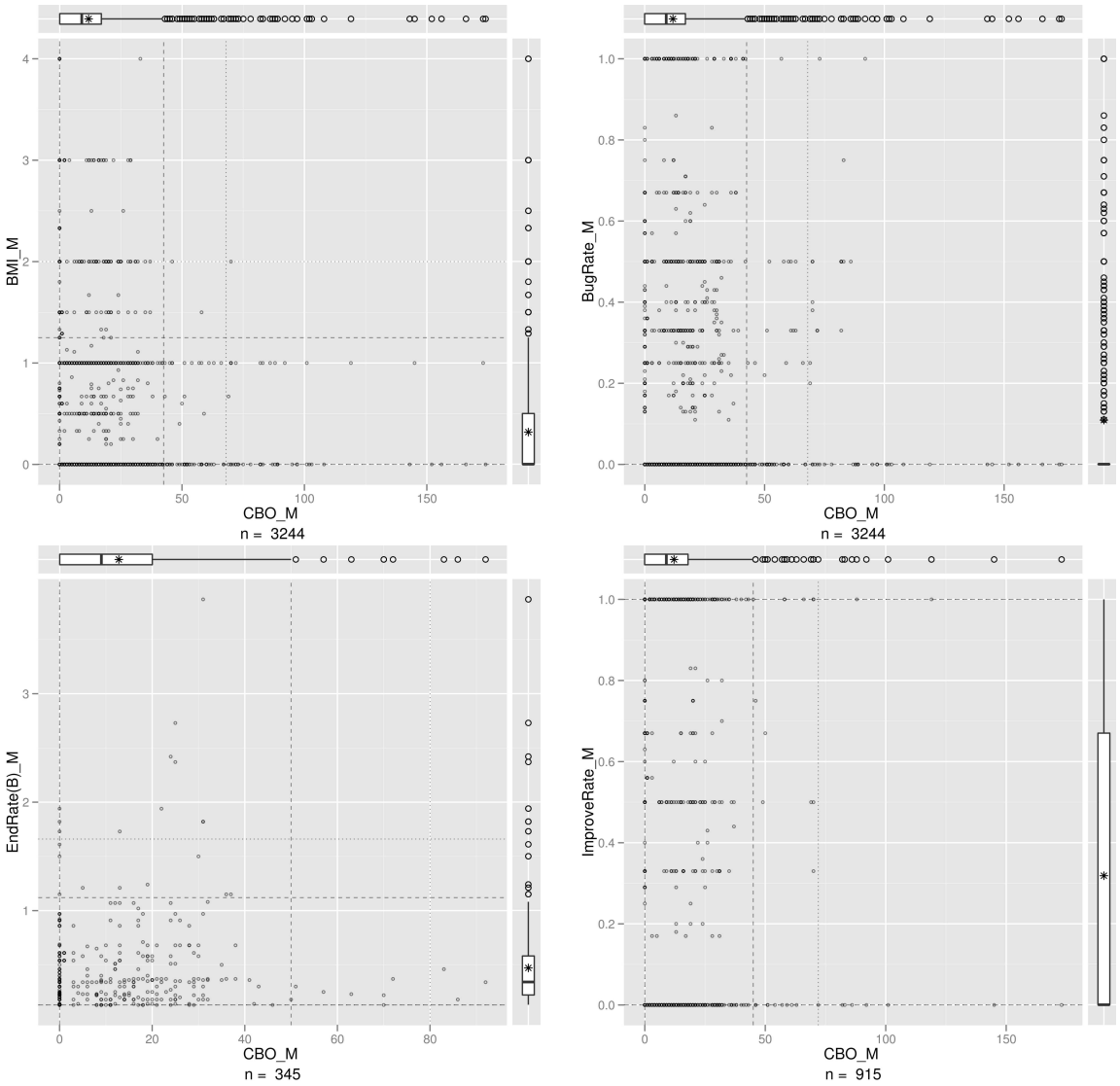


Abbildung D.2 – Streudiagramme für das Maß $\mathrm{CBO_M}$ mit den Maßen (von links nach rechts und von oben nach unten): $\mathrm{BMI_M}$, $\mathrm{BugRate_M}$, $\mathrm{EndRate_M}$, $\mathrm{ImproveRate_M}$

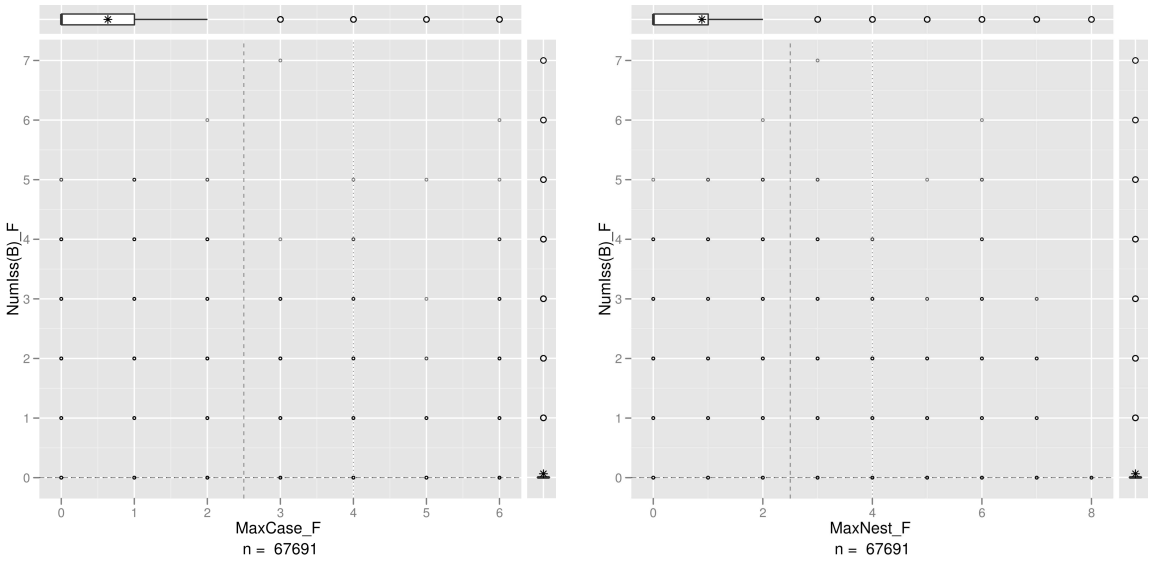


Abbildung D.3 – Streudiagramme für MaxCase$_F$ und NumIss(B)$_F$ (links) sowie MaxNest$_F$ und NumIss(B)$_F$ (rechts)

Index

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe. Weiterhin erkläre ich, eine Diplomarbeit in diesem Studiengebiet erstmalig einzureichen.

Berlin, den 10. Mai 2012